UBC OKANAGAN

COSC 121
Computer Programming II

Recursion

Part 2/2

Dr. Mostafa Mohamed

Acknowledgement: The slides mainly rely on the textbook of Y. D. Liang titled “Introduction to Java Programming, 10th Ed.”, Pearson Edu. Inc.

Outline

Previously:

- Introduction to Recursion
- Think Reclusively
- Examples: Simple Recursive Methods

Today:

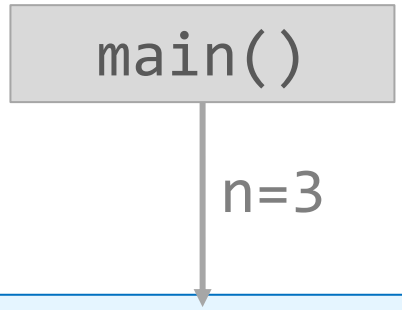
- How Recursion Really Works
- Recursive Helper Methods
- More example problems
  - Ones that have a nice recursive solution
  - Ones that really need recursion
- Tail-Recursive Method
- Recursion vs. Iteration

How Recursion Really Works

Recursion and Memory

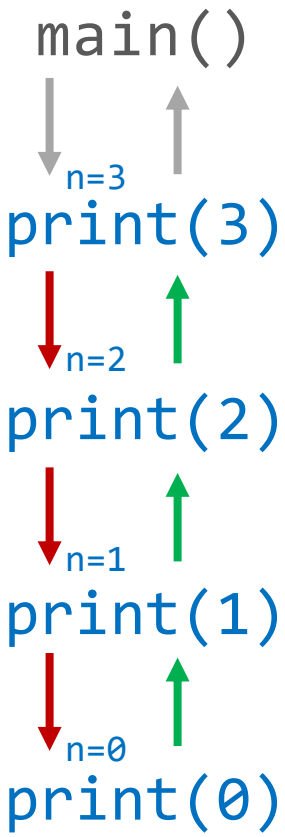
Example 1, *again*

As we saw earlier, this method prints the numbers from n to 1




```
void print(int n){
   if(n > 0) {
      System.out.print(n+" ");
      print(n-1);
   }
   return;
}
```

Let's see what happens if we swap these two statements

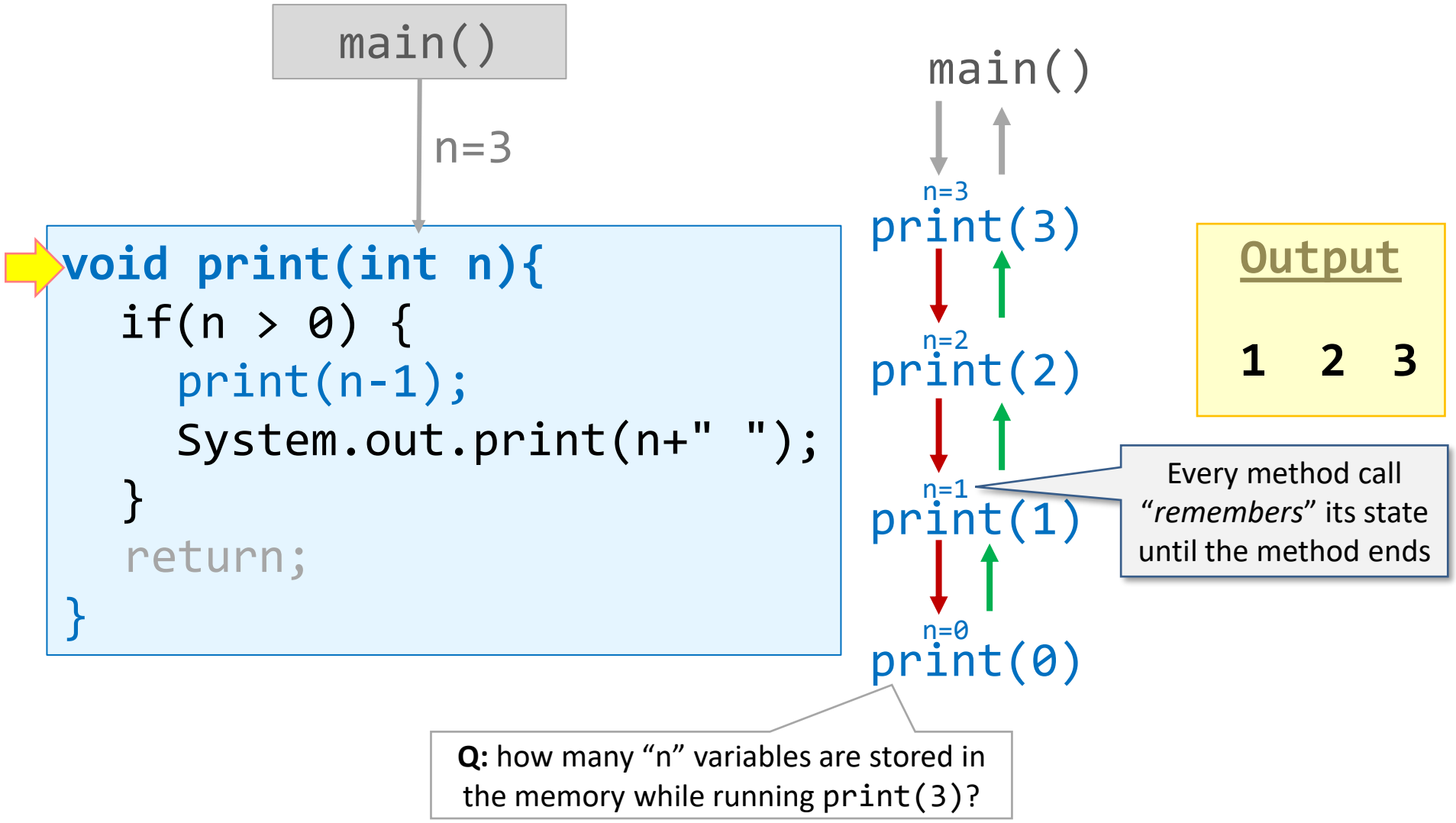


Output

3 2 1

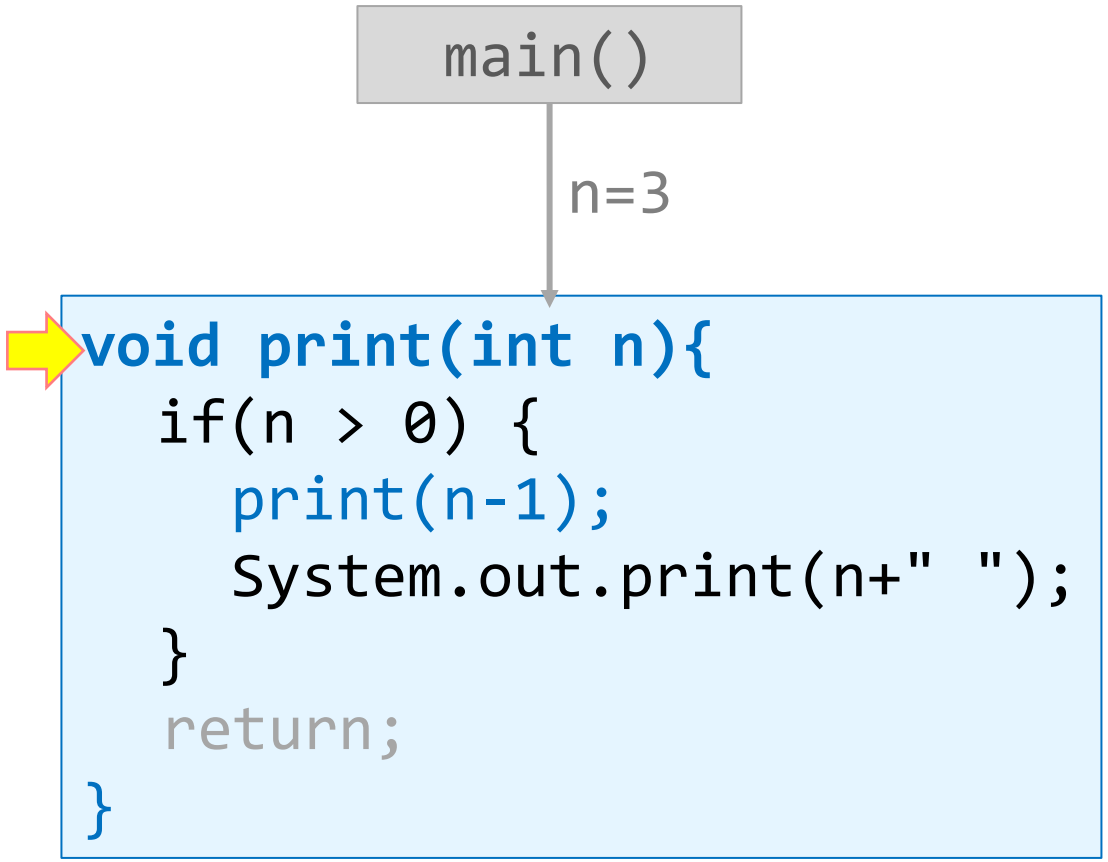
Example 1 (reordered)

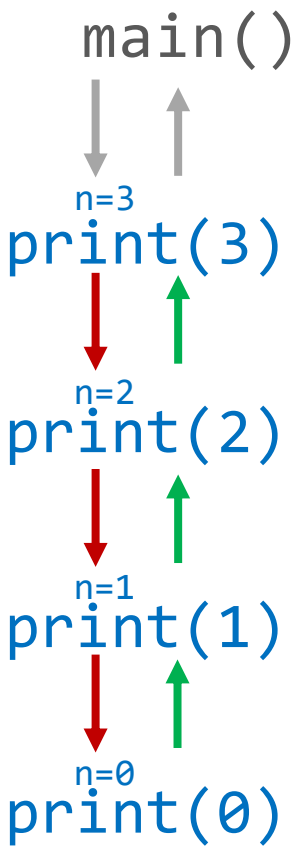
What is the output after reordering the statements?



Example 1 (reordered): *Memory visualization*

Let's see this again with **memory visualization**





Output

1 2 3

Stack

- space for *print(0)* — *local n = 0*
- space for *print(1)* — *local n = 1*
- space for *print(2)* — *local n = 2*
- space for *print(3)* — *local n = 3*
- memory space for *main*

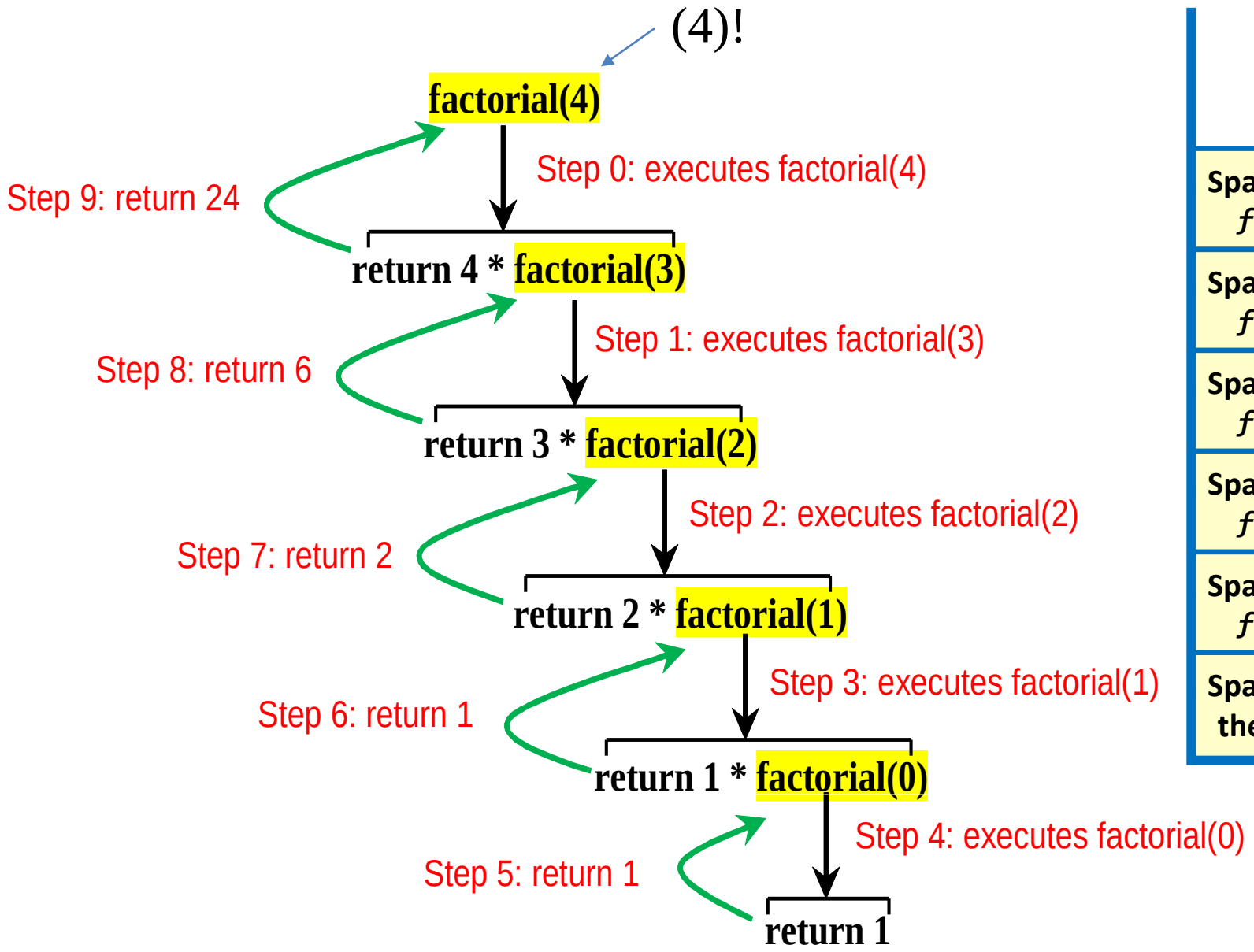
Example 2: Factorial, *again*

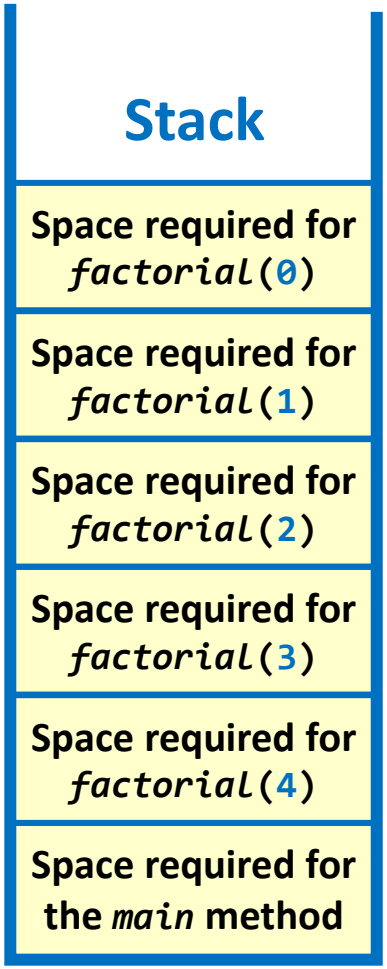
The code:

```
public class Factorial {
    public static void main(String[] args) {
        int f = factorial(4);
        System.out.println(f);
    }

    // Recursive method
    public static int factorial(int n){
        if(n == 0)                          //stopping condition
            return 1;
        else
            return n * factorial(n-1);  //recursive call
    }
}
```

Output: 24

Example 2: Factorial, *Memory Visualization*





Recursive Helper Methods

Recursive Helper Methods

Recursive Helper Method

- A recursive method that is called by another method that is usually non-recursive.

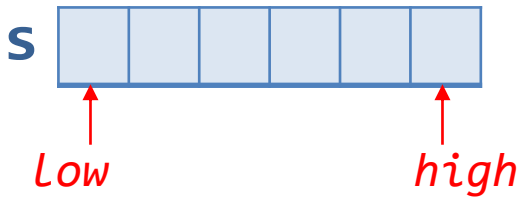
Example:

- The recursive isPalindrome method we saw before is **not efficient**
  - because it creates a new string for every recursive call.
- To avoid creating new strings, use two additional parameters, low and high, that point to the beginning and end of the 'same' string.
  - The two parameters will point to characters from both ends and moving towards the middle (*see the next slide*):
    - **Stopping condition 1**: pointers reach the middle of the word
    - **Stopping condition 2**: the two characters being pointed at are different
    - **Problem**: check characters at end + repeat the string without outer characters.
    - **Recursive call**: check the substring after moving the pointers towards the middle

Recursive Helper Methods, cont'd

```
public static void main(String[] args) {
    System.out.println(isPalindrome("ubc", 0, 2));  //false
    System.out.println(isPalindrome("xyx", 0, 2));  //true
    System.out.println(isPalindrome("x"  , 0, 0));  //true
}
```

```
public static boolean isPalindrome(String s, int low, int high){
    if(low == high)                                 //1)one letter word
        return true;
    else if(s.charAt(low)!=s.charAt(high)) //2)chars at ends diff
        return false;
    else                                            //recursive call
        return isPalindrome(s, low+1, high-1);
}
```



Recursive Helper Methods, cont'd

Example, cont'd

- In the previous slide, you noticed that every time we want to use `isPalindrome`, you have to pass two additional arguments, 1 and length-1.

- This is tedious and could be easily solved by adding an additional non-recursive method that calls our recursive helper method as in the following slide.

Recursive Helper Methods, cont'd

```
public static void main(String[] args) {
    System.out.println(isPalindrome("ubc")); //false
    System.out.println(isPalindrome("xyx")); //true
    System.out.println(isPalindrome("x"));   //true
}
public static boolean isPalindrome(String s){
    return isPalindrome(String s, 0, s.length()-1){
}
public static boolean isPalindrome(String s, int low, int high){
    if(low == high)                          //1)one letter word
        return true;
    else if(s.charAt(low)!=s.charAt(high)) //2)chars at ends diff
        return false;
    else                                     //recursive call
        return isPalindrome(s, low+1, high-1);
}
```

Exercises

Exercise 18.4: Write a recursive method to compute the following series and then test your code to show m(10), m(5), m(1)

$$m(i) = \frac{1}{2} + \frac{2}{3} + \frac{3}{4} + \cdots + \frac{i}{i+1}$$

Exercise 18.10: Write a recursive method that finds the number of occurrences of a specified letter in a string using the following method header. Use a helper method to improve your code. Write a test program to test your method.

```
public static int count(String str, char a)
```

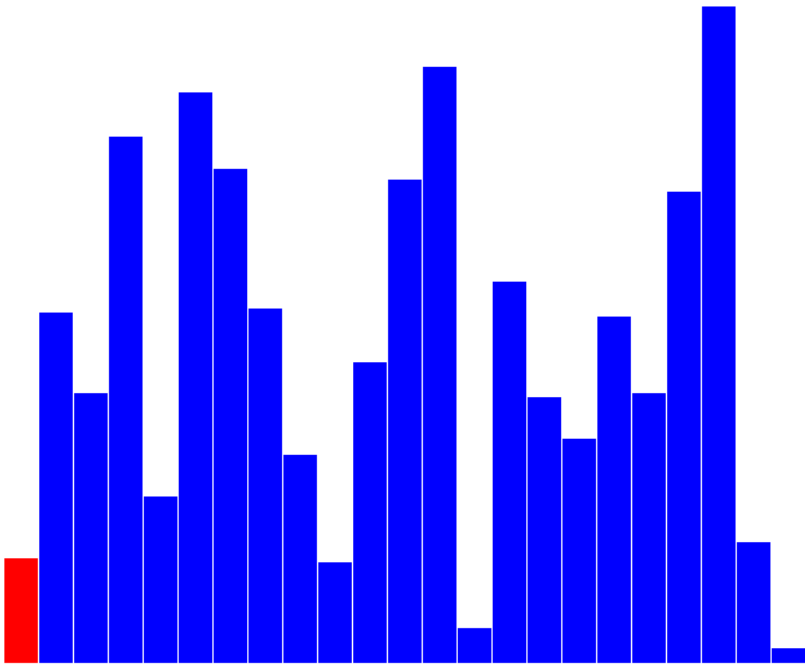
More exercises: Textbook Exercises: 18.1-18.17, 18.28, 18.29.

Challenge question: Textbook Exercise 18.26 (you just need to come-up with the algorithm, but don't worry about building the GUI)

Some Problems that have a nice recursive solution

Recursive Selection Sort

Given a list, sort the items in the list in ascending order.



darcy.rsgc.on.ca

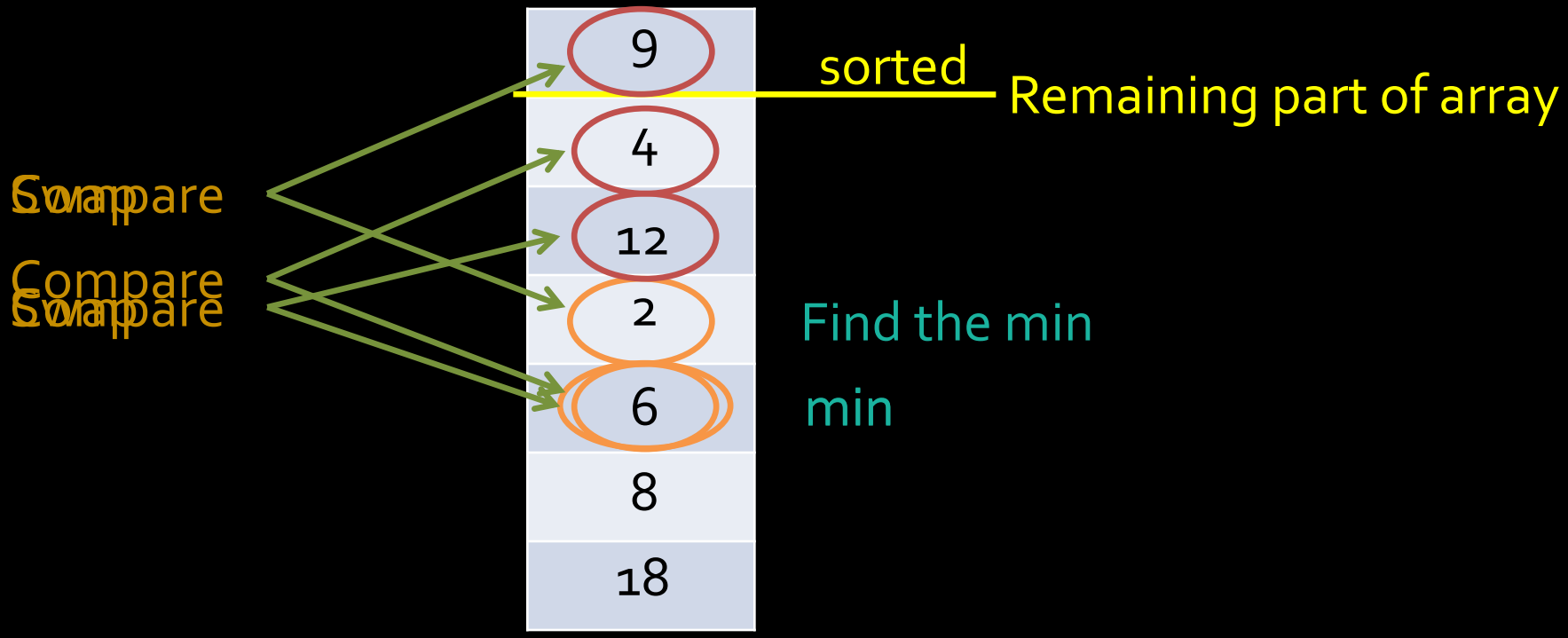
Note that we are repeating the same thing on a smaller array every time

Algorithm:

- Find the smallest element in the list and swap it with the first element.
- Ignore the first element and sort the **remaining smaller list.**

- **Recursion:**
  - **Problem: Sort list** = (put min element first) THEN (sort smaller list).
  - **stopping condition (Base case)**: the list contains only one element.

Recursive selection sort



THE ALGORITHM

As long there is a “remaining part”, do the following:

- Find the smallest element in the remaining part
- Swap with current element
- Recursively repeat the process after ignoring the current element

Recursive Selection Sort

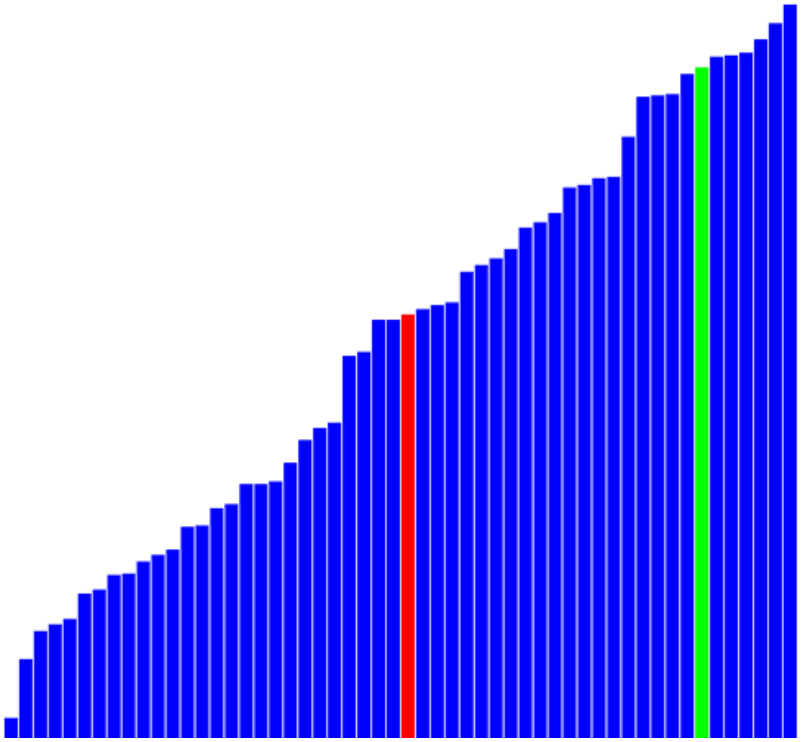
```
public static void main(String[] args) {
    int[] list = { 3, 4, 7, 2, 9 , 1 };
    sort(list);
    System.out.println(Arrays.toString(list)); // list should be sorted
}
private static void sort(int[] list) {
    sort(list, 0, list.length - 1);
}
private static void sort(int[] list, int low, int high) {
    if(low<high) {
        //find the smallest value in list[low ... high]
        int idxMin = low;
        int min = list[low];
        for(int i = low; i<=high; i++) {
            if(list[i] < min) {
                min = list[i];
                idxMin = i;
            }
        }
        //swap min with list[low]
        list[idxMin] = list[low];
        list[low] = min;
        //recursively sort remaining list[low+1, high]
        sort(list, low+1, high);
    }//else if (low == high) return; //stopping condition when list has one element only
}
```

Recursive Binary Search

Algorithm:

Compare the value with the element in the middle of the array:

1. If (value < middle element), recursively search for the key in the first half of the array.
2. If (value > middle element), recursively search for the key in the second half of the array.
3. If (value == middle element), search ends with a match.



darcy.rsgc.on.ca

Recursive calls: Cases 1 and 2 reduce the search to a smaller list.

Stopping conditions:

- A match is found (Case 3 above)
- Cannot find a match (search is exhausted without a match).

Note: *for binary search to work, the array must be sorted.*

Recursive Binary Search

```
public static void main(String[] args) {
    int[] list = {1, 5, 6, 8, 9, 11, 20};
    int key = 7;
    int idx = binarySearch(list, key);   //list must be sorted
    System.out.println(idx);
}

private static int binarySearch(int[] list, int key) {
    return binarySearch(list, key, 0, list.length-1);
}

private static int binarySearch(int[] list, int key, int low, int high) {
    int mid = (low+high)/2;
    if(key == list[mid])     //stopping cond. 1: key found
        return mid;
    else if(low>high)        //stopping cond. 1: key can't be found
        return -1;
    else if(key<list[mid])   //recursive call with smaller list
        return binarySearch(list, key, low, mid-1);
    else
        return binarySearch(list, key, mid+1, high);
}
```

Some problems that REALLY need recursion

More Recursive Problems

The preceding examples can easily be solved without using recursion.

You will see now examples of some problems that are hard to solve without recursion.

- Directory Size
- Tower of Hanoi
- Fractals

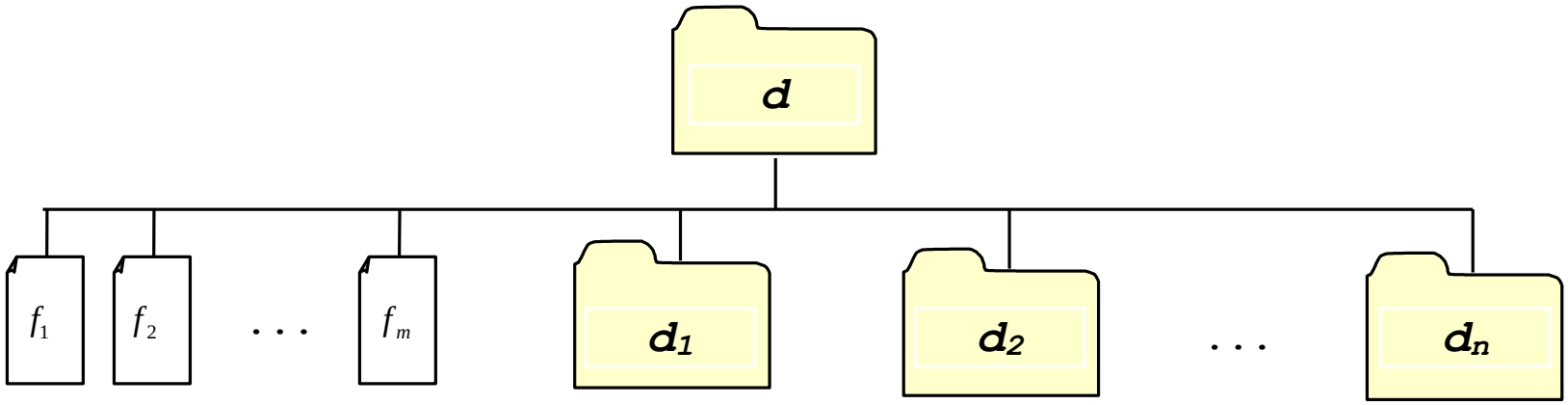
Recursive methods are efficient for solving problems with recursive structures.

Compute Directory Size

Objective: find the size of a directory.

directory size = sum of all files sizes + all subdirectories' sizes

$$size(d) = size(f_1) + size(f_2) + ... + size(f_m) + size(d_1) + size(d_2) + ... + size(d_n)$$



- Useful Java methods (of the `File` class):
  - **`length()`** returns the size of a file in bytes as `long` number
  - **`listFiles()`** returns an array of **`File`** objects under a directory.
  - **`isDirectory()`** returns **`true`** if the **`File`** object refers to a directory

Compute Directory Size

Think of this problem as a tree (branch = folder, leaf = file)

- **Stopping condition**: you reach a leaf (a file: return the size of this file)
- **Recursion**: you find a branch (a directory: return the sum of sizes of all subdirectories and files)
  - size(d) = size($item_1$) + size($item_2$) + … + size($item_n$)
    - item could be a file or a directory

```
public static void main(String[] args) {
    long size = getSize(new File("c:/Dell/"));
    System.out.println(size);
}
private static long getSize(File file) {
    if(file.isFile())
        return file.length();
    else {
        long sum = 0;
        for(File f: file.listFiles())
            sum += getSize(f);
        return sum;
    }
}
```

Practice 6

Write a recursive method that returns the number of all occurrences of a given word in all the files under a directory. Use the following header: **long countInFolder(File f, String word)**

Write a test program.

Idea:

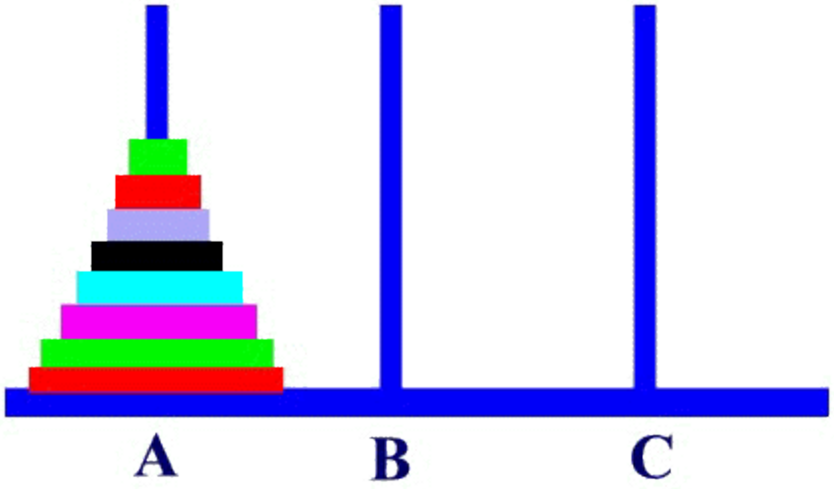
1. Start by creating a non-recursive method `countInFile(File file, String word)`. This method returns the number of occurrences of word in file. Use a `Scanner.next()` to read words from the file and check them.
2. Similar to `getSize()` from previous example, create a recursive method `countInFolder()` that returns the count of a word in that file (as opposed to the size of a folder). i.e.
   - In the countInFolder: if f is a file, then return countInFile(f,word). Otherwise, if f is a folder then recursively apply countInFolder on each item in f.

```
public static void main(String[] args) {
    System.out.println(countInFolder(new File("c:/2/"), "UBC"));
}

public static long countInFolder(File folder, String word) {
    if (folder.isFile())
        return countInFile(folder, word);
    else {
        long count = 0;
        for (File f: folder.listFiles())
            count += countInFolder(f, word);
        return count;
    }
}

public static int countInFile(File file, String word) {
    int count = 0;
    try (Scanner input = new Scanner(file)) {
        while (input.hasNext())
            if (input.next().contains(word))
                count++;
    } catch (Exception ex) {ex.printStackTrace();}
    return count;
}
```

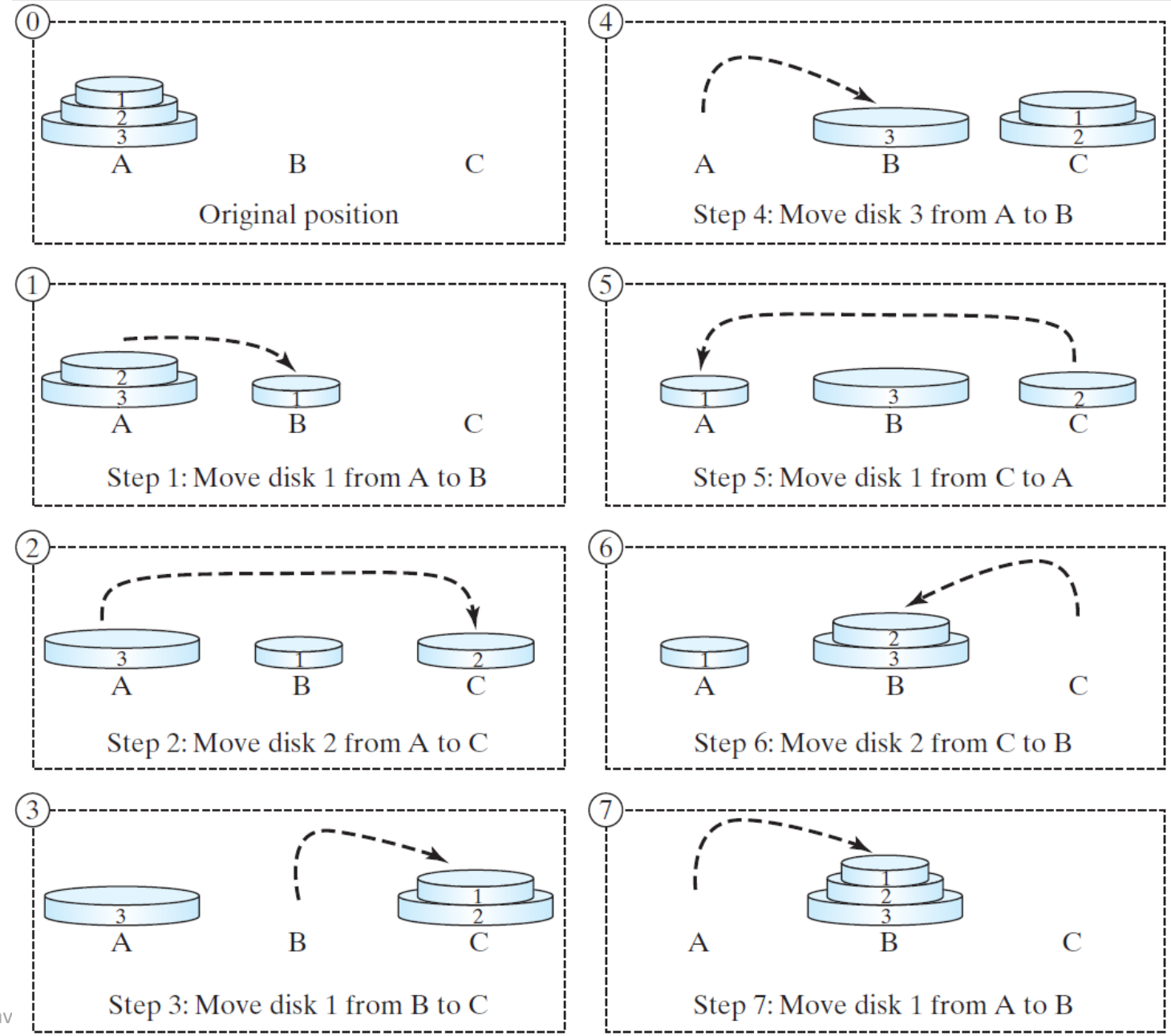
Tower of Hanoi

Problem:

- There are n disks labeled 1, 2, 3, . . ., n, and three towers labeled A, B, and C.
- All the disks are initially placed **on tower A**.
- It is required to move all disks **to tower B** with the assistance of C while observing the following rules:
  - No disk can be on top of a smaller disk at any time.
  - Only one disk can be moved at a time, and it must be the top disk on the tower.



Tower of Hanoi for 3 Discs



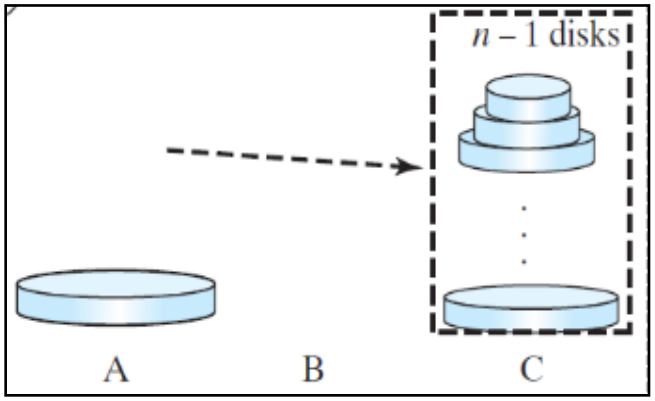
Solution to Tower of Hanoi

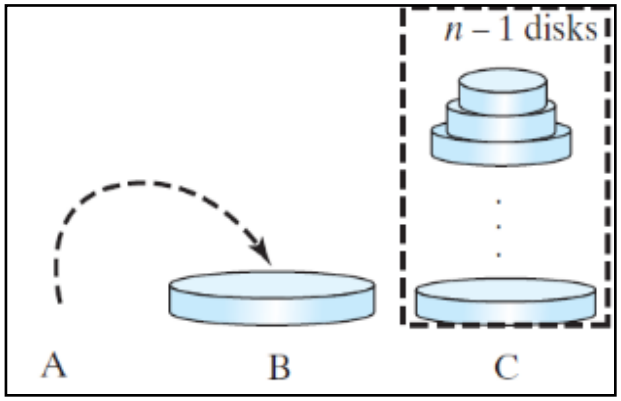
Base case (**n = 1**).

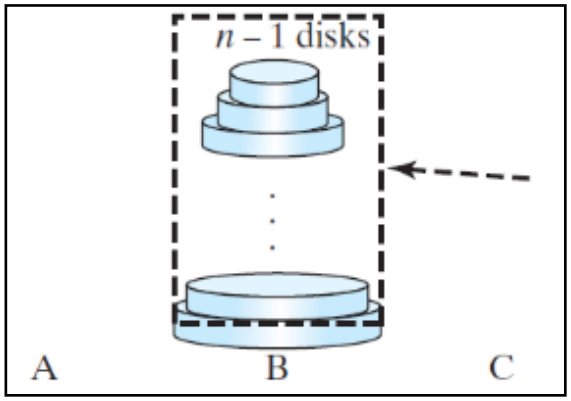
- If **n == 1**, simply move the disk from A to B.

Recursion (**n > 1**):

- split the original problem into three subproblems and solve them sequentially:
  1. Move the first **n - 1** disks from A to C recursively with the assistance of B
  2. Move disk **n** from A to B
  3. Move **n - 1** disks from C to B recursively with the assistance of A

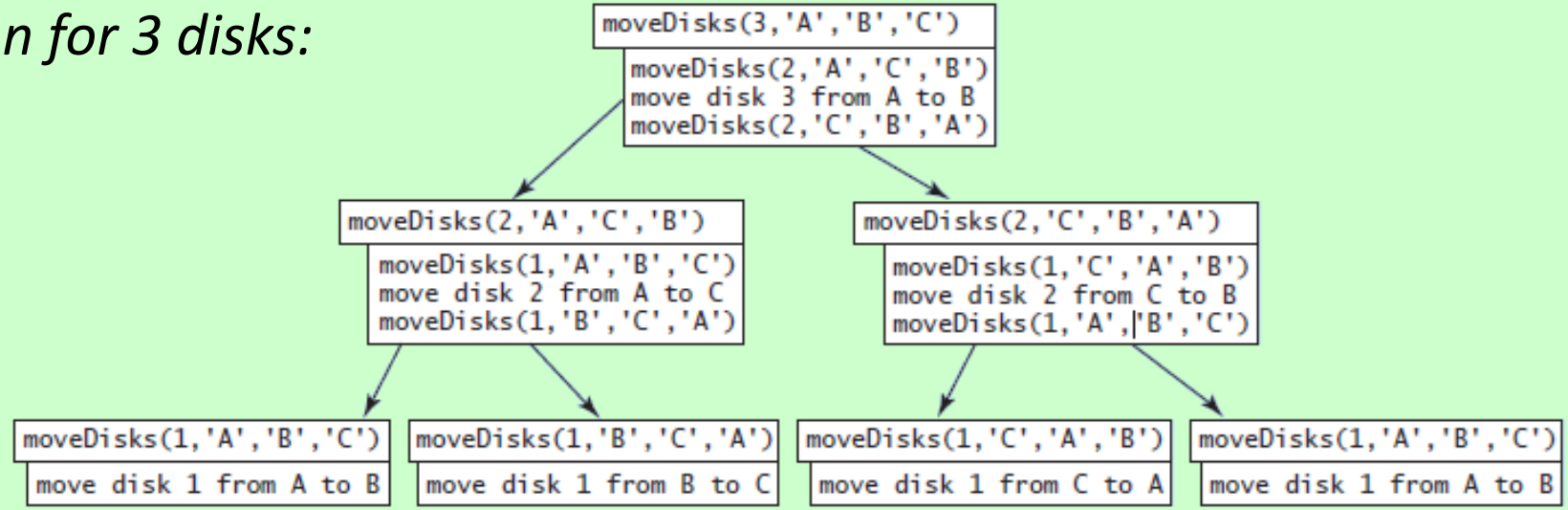






Solution to Tower of Hanoi

Illustration for 3 disks:

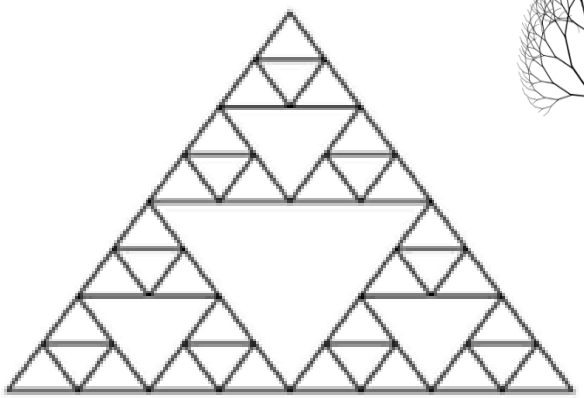


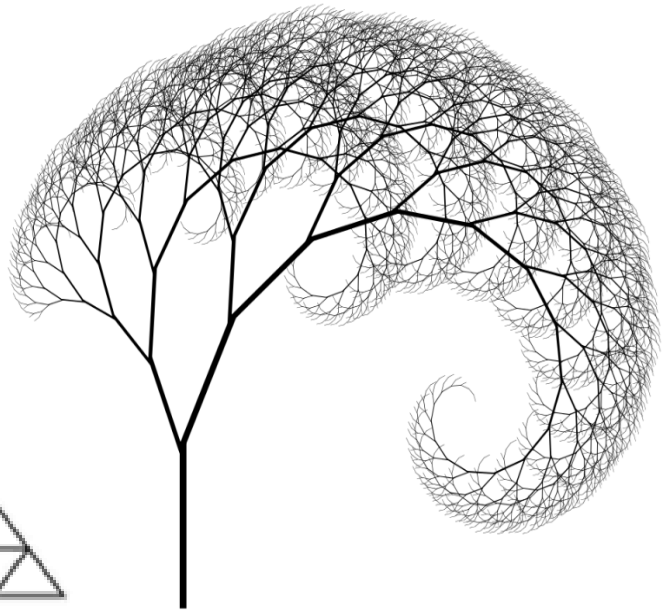

```java
public class TowerOfHanoi {
    public static void main(String[] args) {
        int n = 3;
        moveDisks(n, 'A', 'B', 'C');    //move n disks from A to B with help of C
    }
    private static void moveDisks(int n, char A, char B, char C) {
        if(n==1)    //stopping condition
            System.out.printf("move %d from %c to %c\n", n, A, B);
        else{
            moveDisks(n-1, A, C, B);    //move n-1 disks from A to C with help of B
            System.out.printf("move %d from %c to %c\n", n, A, B);
            moveDisks(n-1, C, B, A);    //move n-1 disks from C to B with help of A
        }
    }
}
```

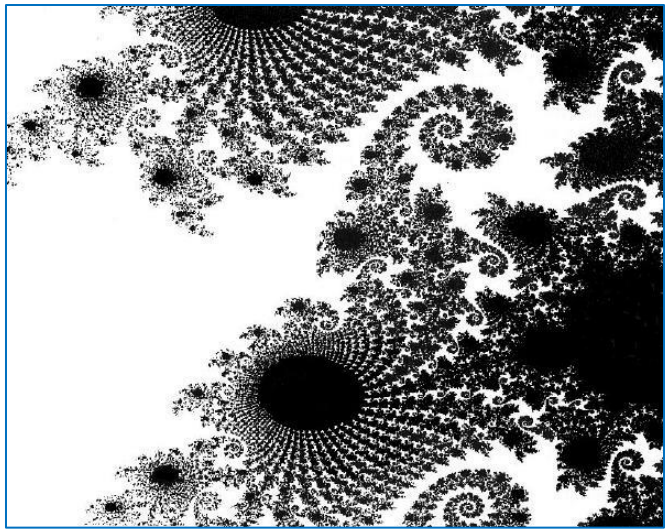
Fractals

After you have learned about recursion in Java, can you think of recursive methods to draw the fractals below?

- Drawing in Java is not covered in this course!



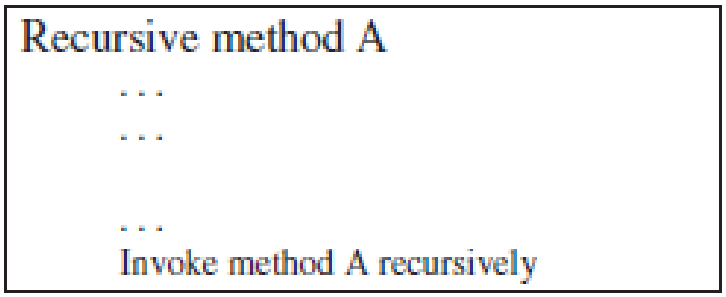




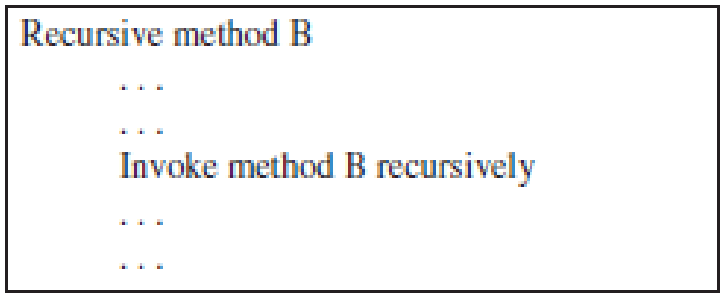
Tail-Recursive Method

Tail-Recursive Methods

A tail-recursive method is the one that has no pending operations to be performed on return from a recursive call.



(a) Tail recursion



(b) Nontail recursion

Tail recursive methods are desirable.

- Compilers can optimize tail recursion to reduce stack size.
  - because the method ends when the last recursive call ends, there is no need to store the intermediate calls in the stack.

A nontail-recursive method can often be converted to a tail-recursive method.

- *We will not discuss how to do that in this course.*

Recursion vs. Iteration

Recursion vs. Iteration

Negative aspects of recursion:

- Recursion bears **substantial overhead**.
  - Each time the program calls a method, the system must assign space for all of the method's local variables and parameters.
  - This can consume **considerable memory** and may require extra time to manage the additional space.

Positive aspects of recursion:

- Using recursion may provide a clear, simple solution for an inherently recursive problem that are difficult to solve without recursion.
  - For example: Directory-size, Tower of Hanoi, and Fractals

When to use which?

- use whichever approach can best develop an intuitive solution that naturally mirrors the problem.
- If an iterative solution is obvious, use it. It will generally be more efficient than the recursive option.

Recursion Efficiency

Let's take a look at Fibonacci again

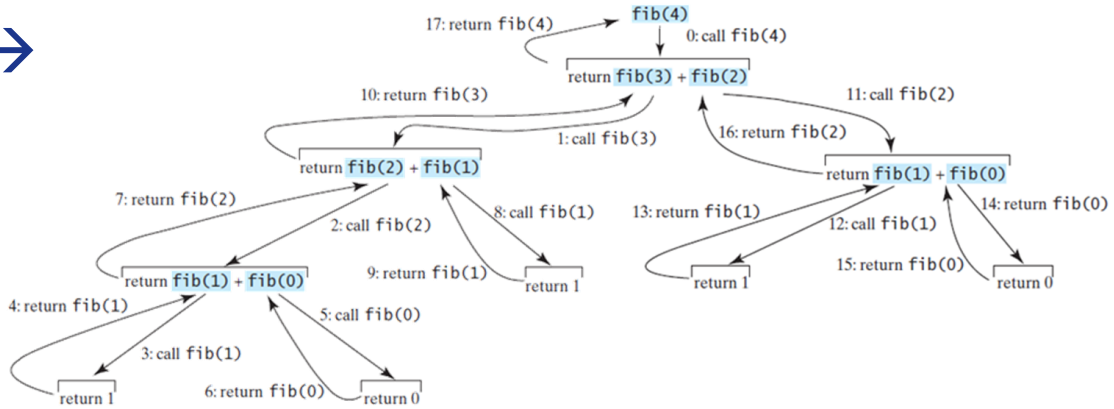
Try the method below with higher index values, e.g. fib(300).

- This will take hours and hours to run.

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

Can you figure out why it takes that long?

- Hint: use this diagram →

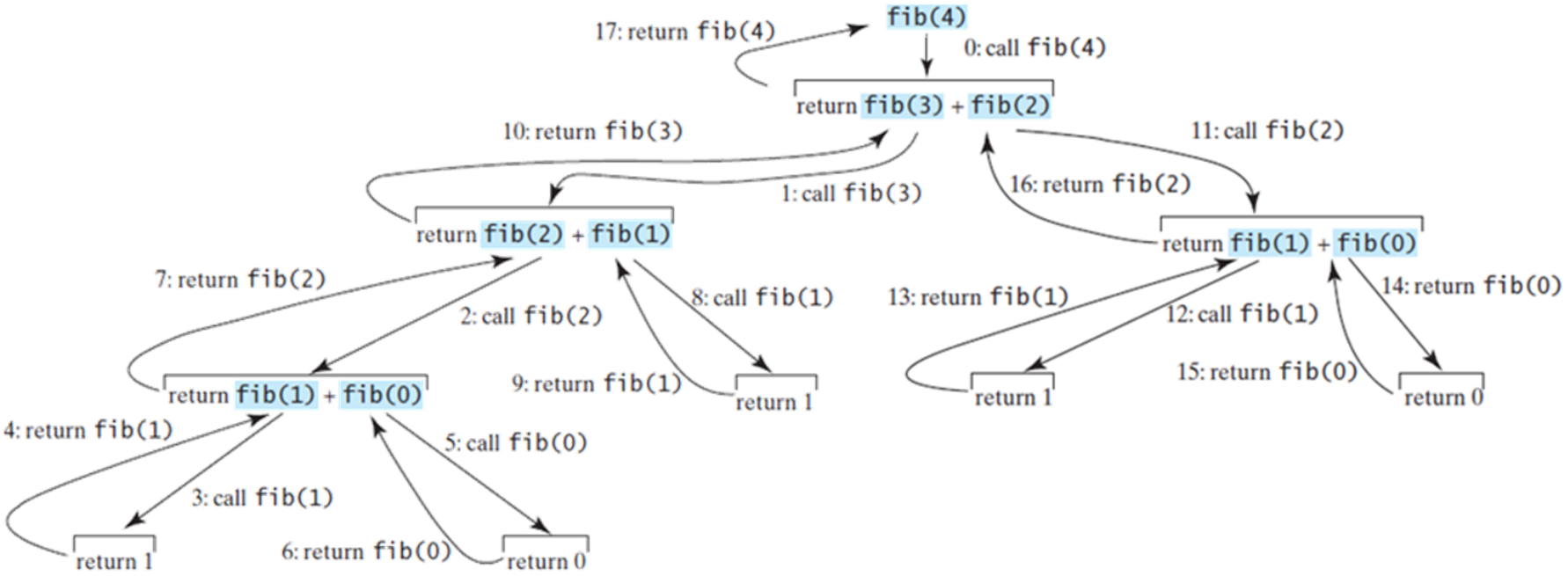


Is there a better way?

Let's take a look at Fibonacci again

In the graph below, you will notice that the Fibonacci for the same index is computed more than once.

- e.g. identify how many times fib(2) is computed



Now imagine a much larger graph – see how many **redundant** calculations we need to perform!!

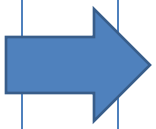
We don’t really need to compute fib(i) more than once! The idea is, once fib(i) is calculated once, we need to store it somewhere then read fib(i) whenever we need it again (instead of re-calculating it).

Here is how we do this. Let’s say that we have a recursive function **F(**parameters**)**:

- 1) Create a global array **M** (outside F) that:
  - Has the same type as the recursive-method return type.
  - Of dimensionality = number of recursive-method parameters.
- 2) before returning a result from F, store it in M at a indexes that match the parameter values of F.
- 3) add one more stopping condition – if M has a value at indexes = F’s parameters, then just return that value.

Solution

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



```
long[] m = new long[1000];

long fib(int i) {
  if(m[i] > 0)
    return m[i];
  if(i == 0 || i == 1)
    return i;
  else {
    m[i] = fib(i-1)+fib(i-2);
    return m[i];
  }
}
```

One last thing…

You might want to use another type for array M since the value can be much larger than what long can hold. A good option is to use BigInteger, a class that can hold very very large integer values.

```
long[] m = new long[1000];

long fib(int i) {
   if(m[i] > 0)
      return m[i];
   if(i == 0 || i == 1)
      return i;
   else {
      m[i] = fib(i-1)+fib(i-2);
      return m[i];
   }
}
```

```
BigInteger[] m = new BigInteger[1000];

BigInteger fib(int i) {
   if(m[idx] != null)
      return m[idx];
   if(i == 0 || i == 1)
      return BigInteger.valueOf(idx);
   else {
      m[idx] = fib(idx-1).add(fib(idx-2));
      return m[idx];
   }
}
```