Chapter 10 – Recursion

Contents

[1 Introduction 2](#_Toc131757120)

[2 Call Stack 3](#_Toc131757121)

[3 Example: Factorial 4](#_Toc131757122)

[4 Examples: Returns Primitive or String, No Helper 6](#_Toc131757123)

[4.1 sumSeries 6](#_Toc131757124)

[4.2 sumInts 7](#_Toc131757125)

[4.3 countChar 7](#_Toc131757126)

[4.4 power 10](#_Toc131757127)

[4.5 isPalindrome 11](#_Toc131757128)

[4.6 Exercises 12](#_Toc131757129)

[5 Examples: Returns Nothing (prints), No Helper 13](#_Toc131757130)

[5.1 recursivePrint 13](#_Toc131757131)

[5.2 printString 15](#_Toc131757132)

[5.3 printReverseString 15](#_Toc131757133)

[5.4 Exercises 15](#_Toc131757134)

[6 Examples: Recursive Helper Methods 16](#_Toc131757135)

[6.1 sumArray 16](#_Toc131757136)

[6.2 countCodeAbc 17](#_Toc131757137)

[6.3 isPalindrone 19](#_Toc131757138)

[6.4 Exercises 20](#_Toc131757139)

[7 Examples: Helper, Store Intermediate Value 21](#_Toc131757140)

[7.1 minInt 21](#_Toc131757141)

[7.2 minString 22](#_Toc131757142)

[7.3 factorial 23](#_Toc131757143)

[7.4 Exercises 23](#_Toc131757144)

[8 Binary Search 24](#_Toc131757145)

[8.1 Example 1 24](#_Toc131757146)

[8.2 Example 2 25](#_Toc131757147)

[8.3 Example 3 26](#_Toc131757148)

[8.4 Development of the Binary Search Algorithm 27](#_Toc131757149)

[8.5 Binary Search implementation for an array of integers. 27](#_Toc131757150)

[8.6 Final Comments on Binary Search 28](#_Toc131757151)

[Appendix 1 Video Lecture on Recursion from MIT 29](#_Toc131757152)

[Appendix 2 Factorial Example 29](#_Toc131757153)

[Appendix 3 Problem: Computing Fibonacci Numbers 29](#_Toc131757154)

[Appendix 4 Selection Sort 30](#_Toc131757155)

[Appendix 5 Tail Recursion 34](#_Toc131757156)

[Appendix 6 Recursion vs. Iteration 35](#_Toc131757157)

[Appendix 7 Problem: printFolder 35](#_Toc131757158)

[Appendix 8 Problem: printFolder with Indent 36](#_Toc131757159)

[Appendix 9 Problem: getFiles 38](#_Toc131757160)

# Introduction

*Recursion* is a problem-solving technique that is another form of repetition (iteration, looping) and is useful for certain types of problems. We use an example here to briefly introduce recursion. In [Section 3](#_Example:_Factorial), we look at the example more carefully. As an example, in mathematics, the factorial of a number is the product of all positive integers from 1 to *.* We use the symbol, , to represent the factorial of *,* which can be written:

We can easily implement this with an iterative algorithm:

**private** **static** **int** factorial(**int** n) {

**long** fact = 1;

**for**(**int** i=2; i<=n; i++) {

fact \*= i;

}

**return** fact;

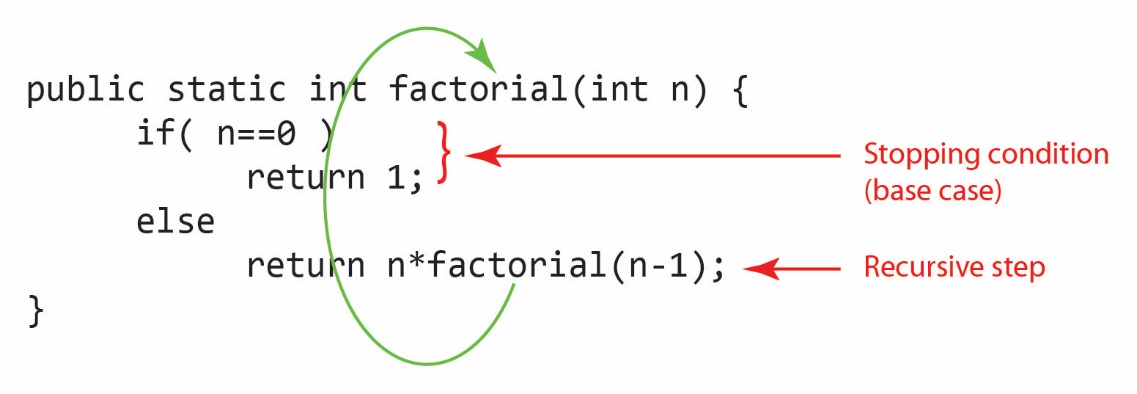
}

Considering the formula for factorial above, we can see that we can represent it with a *recursive function*:

Somewhat loosely, we say that something is *recursive* if it is defined in terms of itself. As we see above, the definition of involves . We say that this is a *recursive definition* of factorial.

In computing, a *recursive method* is a method that calls itself. For example, a recursive implementation of factorial is shown below. Note that:

* There is a *recursive step* where the method calls itself with a smaller value of .
* There is a *stopping condition* (*base case*), otherwise the recursion would continue indefinitely.

****

All recursive methods have the following characteristics:

1. An *if/else* or *switch* statement that leads to different cases.
2. One or more simple cases, called *base cases,* that can be solved without a recursive call. In other words, a base case ends the recursion. Thus, sometimes we refer to these as *stopping conditions*.
3. Every recursive call creates a smaller version of exactly the same problem bringing it increasingly closer to a base case until it becomes that base case (stopping the recursion).

Most of the examples we consider are more naturally solved with a loop; however, we will solve them with recursion. The reason for this is to study how recursion works on simpler problems before considering more complex examples where recursion is the natural choice.

# Call Stack

To understand recursion, it is helpful to understand the *call stack.* A *call stack* is a data structure that stores information about the active methods in a program. When a method is executing there is an *active frame* on the *top* of the stack that represents the state of the method. When that method calls another method, the called method is *pushed* on to the stack and becomes active and the callee becomes inactive. When the method ends, its frame is *popped* from the call stack and the callee becomes active again at the line of code where it left off.

For example, consider this code:

**public** **static** **void** main(String[] args) {

**int** x = 3;

**int** y = 7;

**double** area = *rectangleArea*(x,y);

System.***out***.println(area);

}

**public** **static** **int** rectangleArea(**int** len, **int** wid) {

**int** area = len \* wid;

**return** area;

}

Below, we illustrate how the stack is utilized as the code above executes.

|  |  |  |
| --- | --- | --- |
| 1. Initially, the call stack is empty. |  |  |
| 1. When *main* is called, it is pushed onto the stack and its code executes. |  |  |
| 1. When *rectangleArea* is called, it is pushed onto the stack and its code executes. |  |  |
| 1. When *rectangleArea* is complete, it is popped from the stack and *main* resumes execution where it left off. |  |  |
| 1. Finally, when *main* is complete, it is popped and the stack is empty. |  |  |

A useful tool for visualizing the call stack is [Java Visualizer](https://cscircles.cemc.uwaterloo.ca/java_visualize/). Copy the code above (*main* and *rectangleArea*) inside the class found at the webpage. Then choose: Visualize execution. On the resulting page, continue to choose: Forward >, and watch how the stack (labelled *Frames* on the webpage) is utilized.

# Example: Factorial

As we saw earlier, a recursive implementation of factorial is:

**public** **static** **int** factorial(**int** n) {

**if**(n==0) { // base case (stopping rule)

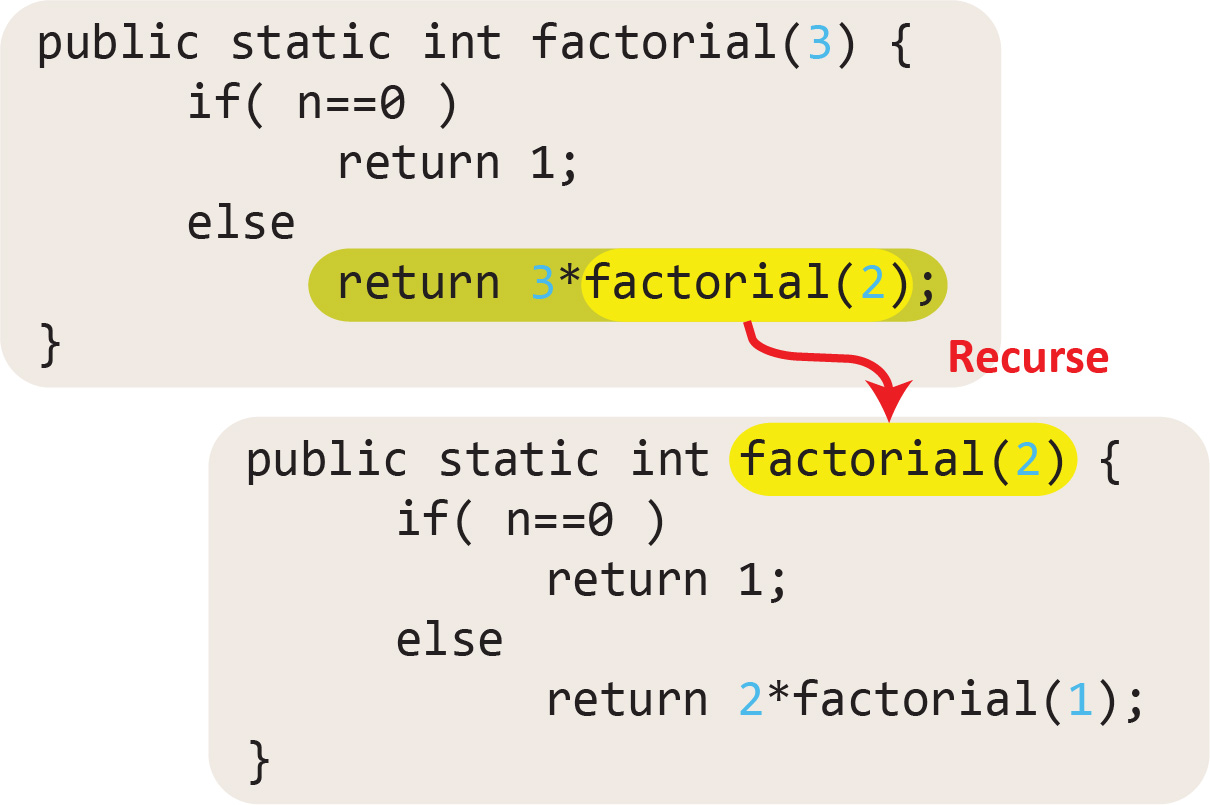
**return** 1;

}

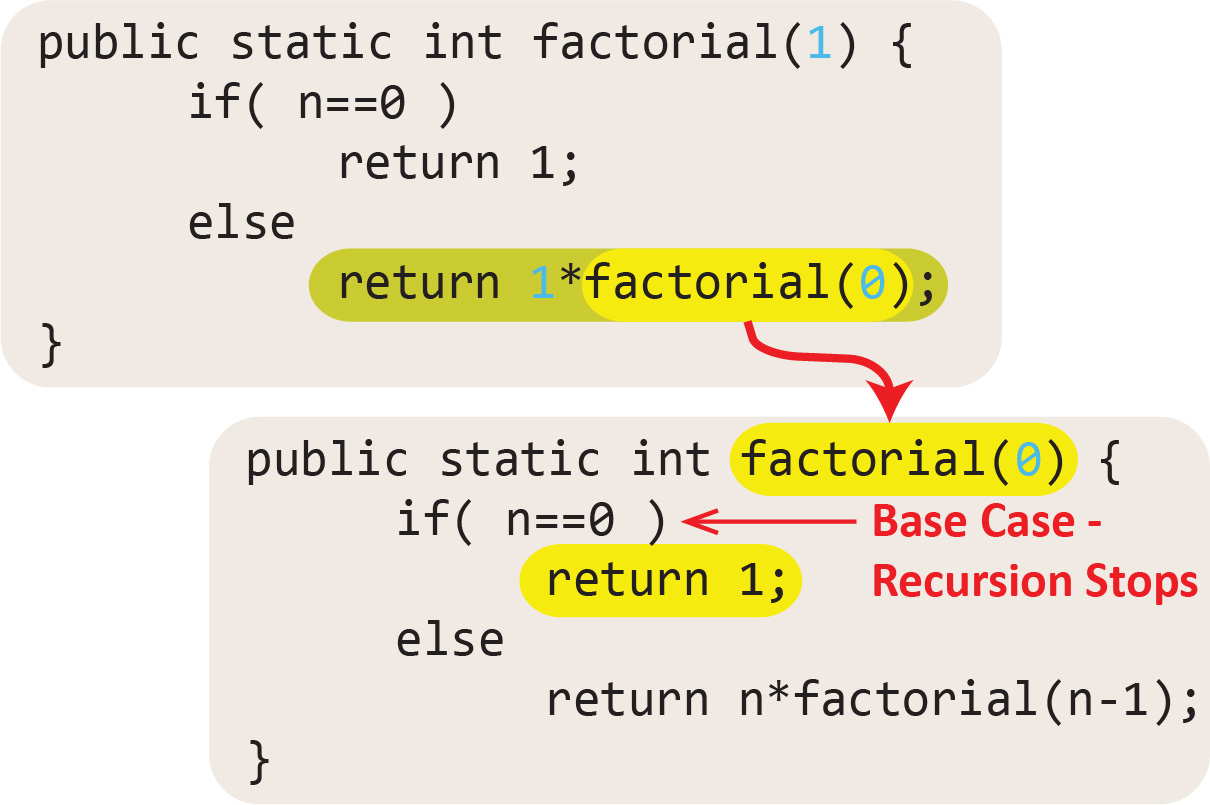
**return** n \* *factorial*(n-1); // recursive step

}

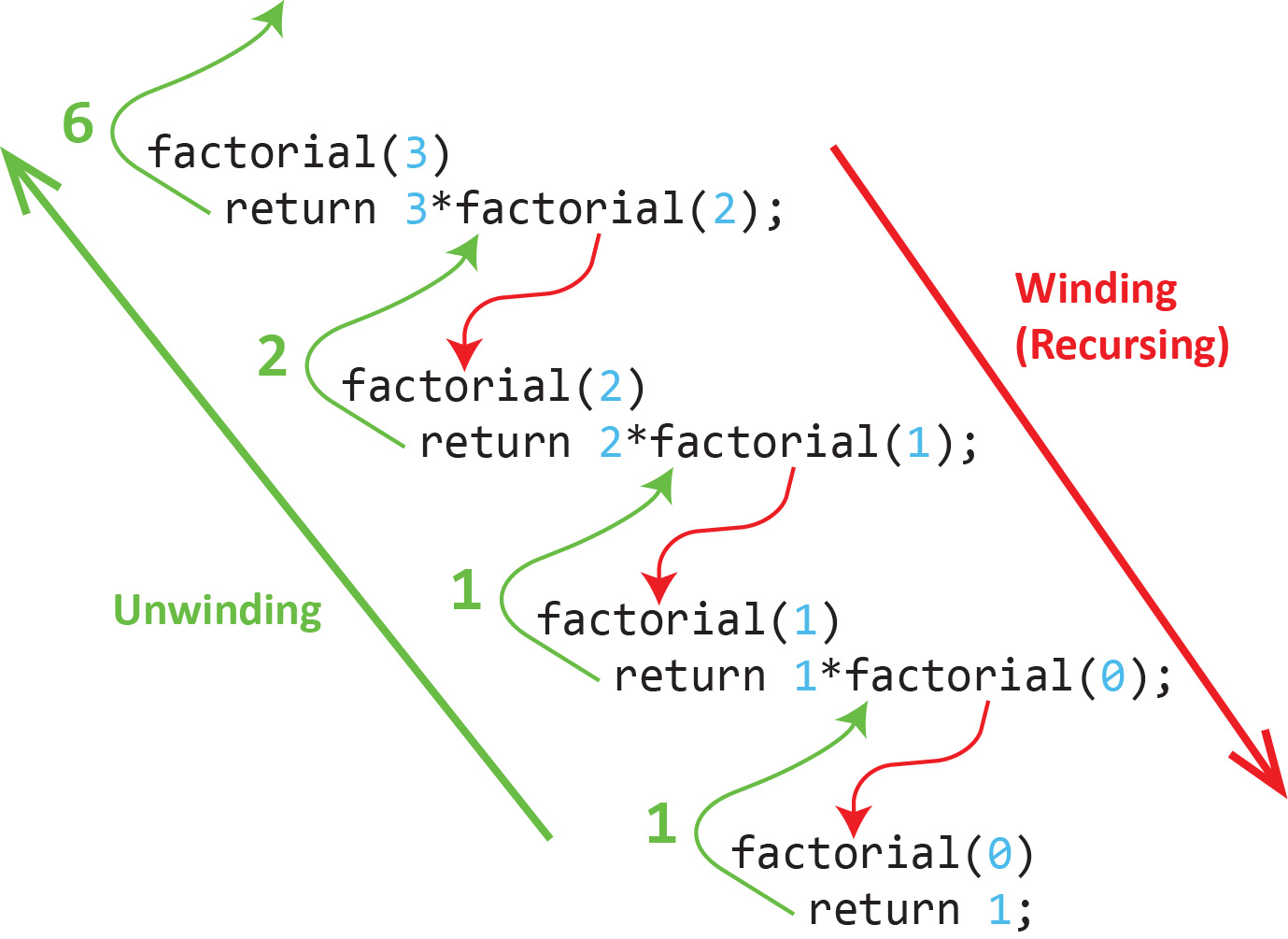
Let’s look more closely at what is happening when, for example, *factorial(3)* is called. Consider the figure below. As we see, in the call to *factorial(3)*, before the return can take place, *factorial(2)* is called. Thus, at some point later, the code execution must come back to the return statement.



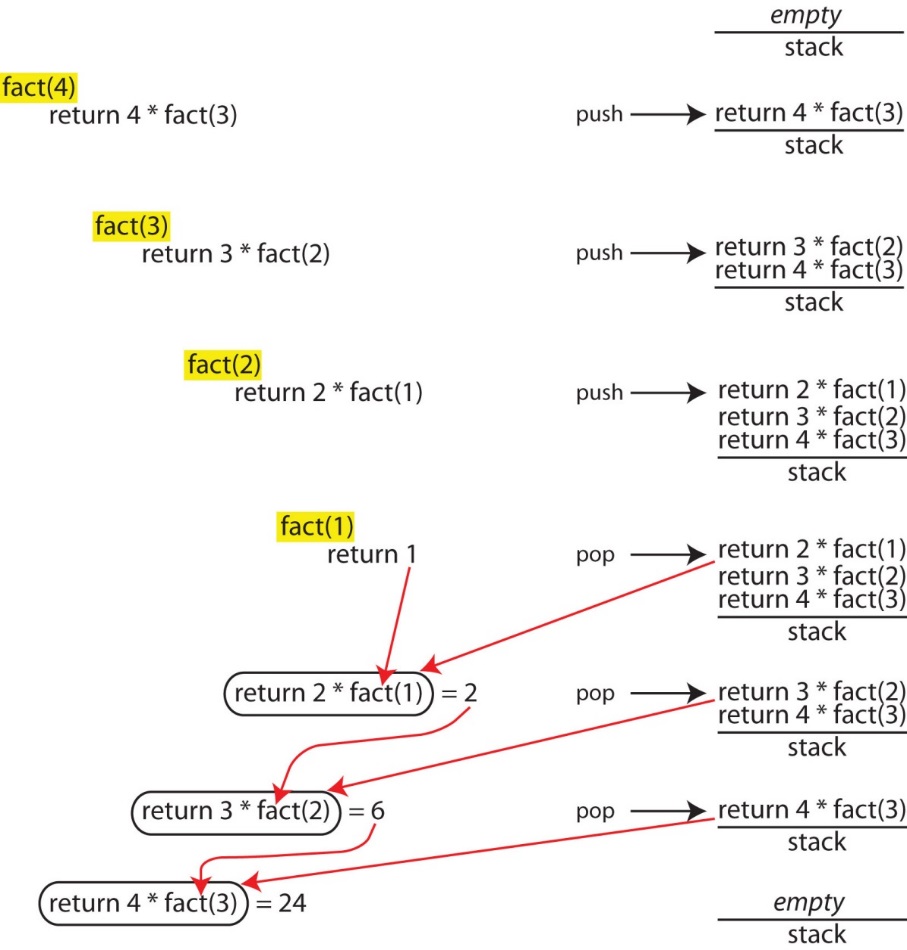
This process continues, eventually, *factorial(0)* is called where the base case stops the recursion by simply returning the value of 1.



The process of recursing until hitting the base case is called *winding* as depicted in the figure below. After the base case, the recursion *unwinds* which means that as each invocation ends, the execution returns to the place it left off in the calling invocation. Notice how the answer is computed as the code unwinds.



Next, let’s visualize how the stack is utilized to calculate *factorial(4).* This is similar to the two sets of figures above, except presented a little more formally. The version of *factorial* we use here (not shown) uses a base case of n=0 or n=1 (which both return 1). Also note that we are using a slightly different visualization of the stack than what is described above; however, the spirit of its use is the same.



Finally, consider what happens if we run the factorial method with no stopping condition:

**public** **static** **int** factorial(**long** n) {

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

}

For example, if this method is run with 4, the result is:

*factorial(4)*

*4\*factorial(3)*

*3\*factorial(2)*

*2\*factorial(1)*

*1\*factorial(0)*

*0\*factorial(-1)* // Not defined

*-1\*factorial(-2)* // Not defined

*...*

If you run the method above, you will get a *StackOverflowError* and the output will be similar to:

Exception in thread "main" java.lang.StackOverflowError

at prob1.FactorialExample.factorial(FactorialExample.java:36)

at prob1.FactorialExample.factorial(FactorialExample.java:36)

at prob1.FactorialExample.factorial(FactorialExample.java:36)

at prob1.FactorialExample.factorial(FactorialExample.java:36)

at prob1.FactorialExample.factorial(FactorialExample.java:36)

...

On my computer (April 5, 2021), the stack overflow error occurred when *n* reached -7432.

# Examples: Returns Primitive or String, No Helper

The examples in this section use recursion to solve problems that return either a primitive or a string and they do not utilize a helper method (we learn what these are shortly). Note also, that the methods we write here are static. That is because they are simply utility methods (*i.e* they do not depend on the state of an object). In the three sections that follow this one, we present more examples that have different characteristics. In other words, for the sake of this course, we are considering four different types of recursion problems, though there are others we don’t consider.

## sumSeries

(Solution in *example\_primitive\_no\_helper package*, *SumSeries* class). Write a recursive method to compute the following series: , for any positive integer.

Of course, this problem is easily solved with a loop; however, we have been asked to solve it recursively. Remember that one of the features of a recursive algorithm is the *recursive step*, where we solve a smaller version of the original problem. Start by listing out some of the first values of the series and we will see the recursive nature of this function:

Thus, another way to represent the function is:

Finally, we can implement a recursive algorithm to solve the problem.

**public** **static** **double** m(**int** n) {

**if**(n<=1) { // base case

**return** 1;

}

**return** *m*(n-1) + 1.0/n; // recursive step

}

## sumInts

(Solution in *example\_primitive\_no\_helper package*, *SumInts* class). Write a recursive method, *sumInts* that accepts an integer and returns the sum of the integers from 1 to the input integer.

It may be hard to see the recursive pattern but consider that it is just like the factorial problem, except adding instead of multiplying. Start by listing a few values of the function and observe the recursive nature:

Thus, another way to represent the function is:

Finally, we can implement a recursive algorithm to solve the problem.

**public** **static** **long** sumInts(**int** n) {

**if**(n==1) { // base case

**return** 1;

}

**return** n + *sumInts*(n-1); // recursive step

}

## countChar

(Solution in *example\_primitive\_no\_helper package*, *CountChar* class). Write a recursive method, *countChar* that accepts a string and a character. The method should return the number of occurrences of the character in the string. For example, count(“tree house”,’e’) returns 3.

First, let’s write the header for the method, noting that we need to return the count (*int*) of how many times *ch* occurs in *str*.:

**public** **static** **int** count(String str, **char** ch) {

Next, let’s look for a recursive pattern. Specifically, let’s see if we can make the problem smaller.

**Idea:** At each step (inside each recursive call):

* Check to see if the first character in *str* is *ch*. If it is, increment a counter
* Then, strip this first character off and recurse with the remainder of the string.

In this way, the string *str* gets shorter with each recursive call. Eventually, we reach the base case, str.length==0.

We haven’t addressed how to “count” in the idea presented above. This method must return an integer on every recursive call. Thus, the answer is simple enough, we just add 1 to the return when we find a match. Thus, the algorithms is:

int countChar(str,ch)

If s.length==0 // base case

return 0

If first character in str is ch // recursive steps

str = remove first character and keep the rest

return 1 + count(str,ch)

Else

str = remove first character and keep the rest

return 0 + count(str,ch)

Finally, the implementation of the algorithm:

**public** **static** **int** countChar(String s, **char** val) {

**if**(s.length()==0) { // base case

**return** 0;

}

// recursive step

**if**(s.charAt(0)==val) {

**return** 1 + *countChar*(s.substring(1),val);

}

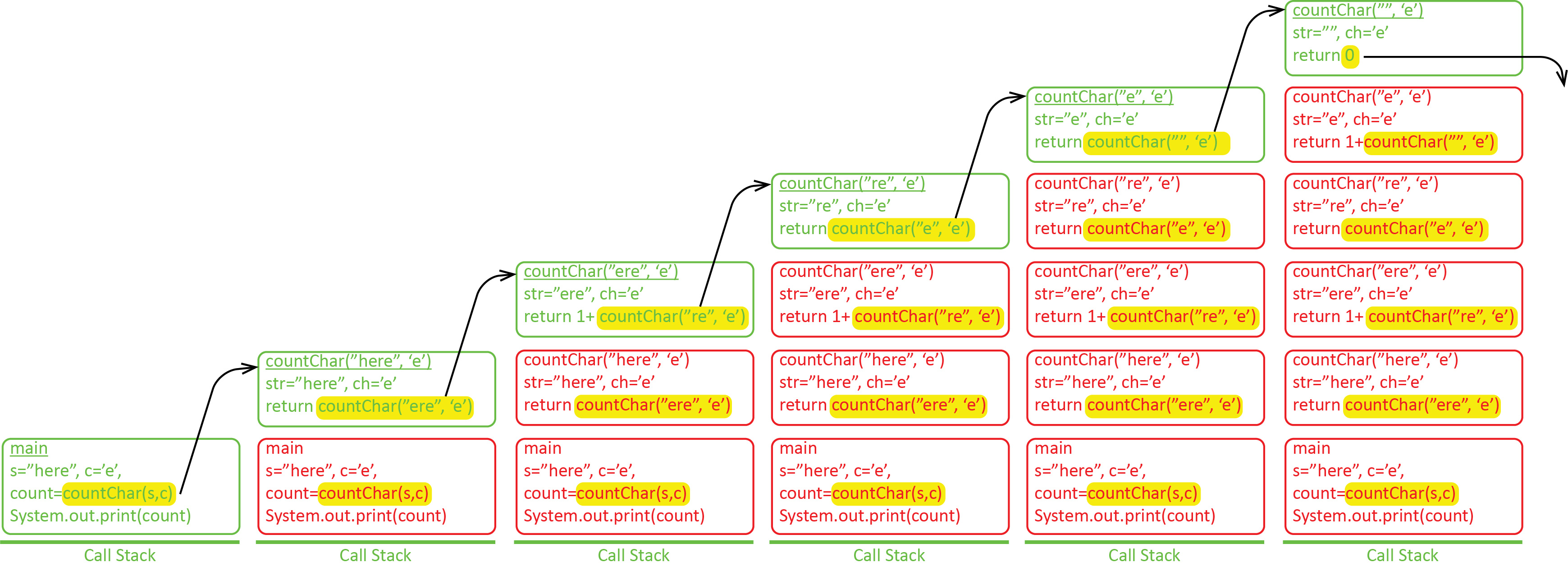
**else** {

**return** *countChar*(s.substring(1),val);

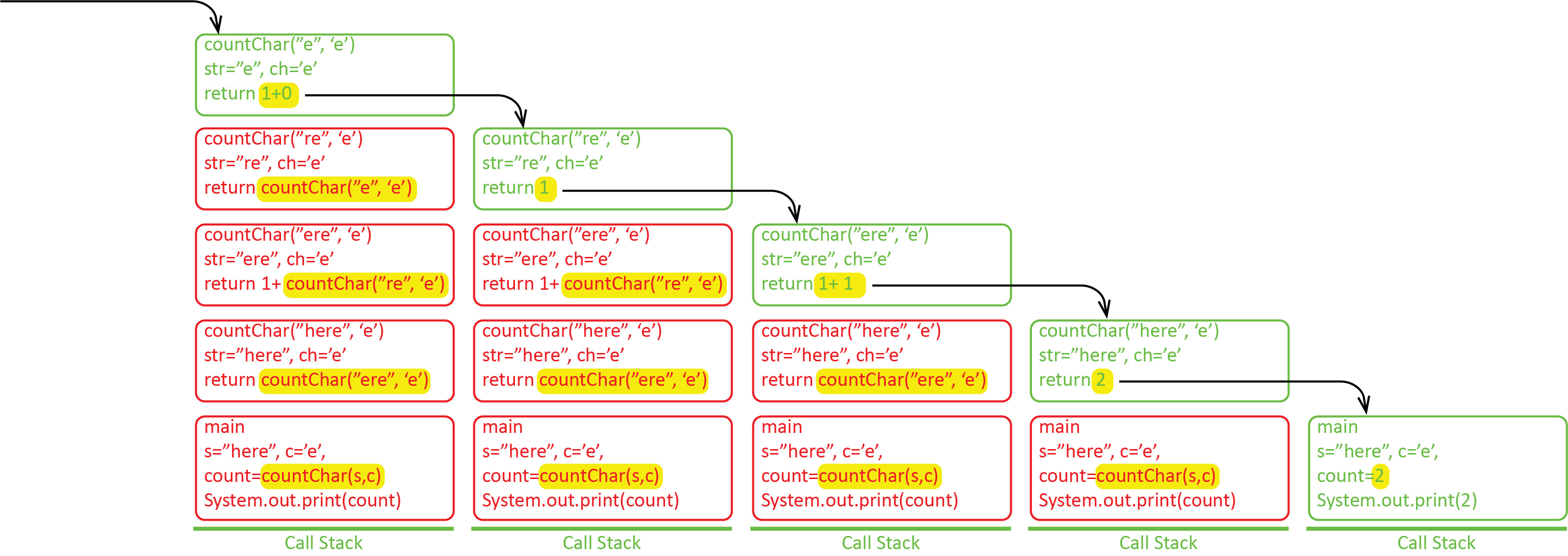
}

}

Consider the stack as *countChar(“here”, ‘e’)* is executed. It begins with *main* calling *countChar* and then the recursion (winding) begins until it hits the base case:



Then the unwinding takes place, popping the stack as each previous invocation ends, accumulating the 1’s and 0’s to arrive at the final answer:



Finally, note that recursive step could be written slightly more simple so that there is only one recursive call:

**public** **static** **int** countChar2(String s, **char** val) {

**if**(s.length()==0) {

**return** 0;

}

**int** count = 0;

**if**(s.charAt(0)==val) {

count=1;

}

**return** count + *countChar2*(s.substring(1),val);

}

Or even shorter, but less readable, using Java’s [ternary, conditional operator](https://www.w3schools.com/java/java_conditions_shorthand.asp):

**private** **static** **int** countChar3(String s, **char** val) {

**if**(s.length()==0) { // base case

**return** 0;

}

**return** (s.charAt(0)==val ? 1 : 0) + *countChar3*(s.substring(1),val);

}

Or, just one line using the ternary operator twice:

**private** **static** **int** countChar4(String s, **char** val) {

**return** (s.length()==0 ? 0 : (s.charAt(0)==val ? 1 : 0) + *countChar4*(s.substring(1),val));

}

## power

(Solution in *example\_primitive\_no\_helper package*, *Power* class). Write a recursive method, *pow(x,n)* that raises *x* to the power of *n, e.g.* .

From basic algebra we know that: . First, let’s find the recursive pattern and the base case:

// base case

Thus, another way to represent the function is:

Finally, we can implement a recursive algorithm to solve the problem.

**public** **static** **double** pow(**double** x, **int** n ) {

**if**(n==0) { // base case (stopping rule)

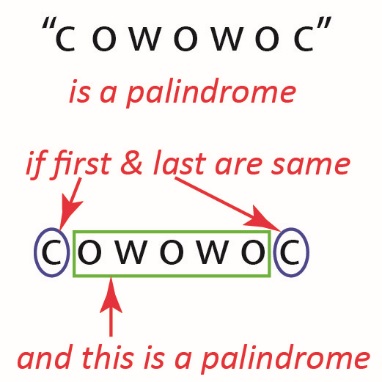
**return** 1;

}

**return** x \* *pow*(x,n-1); // recursive step

}

## isPalindrome

(Solution in *example\_primitive\_no\_helper package*, *IsPalindrome* class). A string is considered a *palindrome* if it reads the same forwards and backwards. Write a method, *isPalindrome* that accepts a string and returns *true* if the string is a palindrome and *false* otherwise.

An idea for a recursive algorithm to solve the problem is shown on the right. We can express this algorithm:

isPalindrome( s )

if length of s is 1 or less // stopping rule 1

return true

else if first and last characters are different // stopping rule 2

return false

else

remove first and last characters from s

return isPalindrone( s )

Consider this example:

isPalindrome( “wobow” )

As the code executes, we make these recursive calls:

isPalindrome( “obo” )

isPalindrome( “b” )

As we can see, the last call will return *true* because the string has length 1, and of course “b” is a palindrome. From there, the recursion unwinds, passing the value of *true* to each previous invocation.

Consider this example:

isPalindrome( “wobxw” )

As the code executes, we make this recursive call:

isPalindrome( “obx” )

At this step, the “o” and “x” are compared and found to not be equal which stops the recursion by returning *false,* whichis passed through the returns as the recursion unwinds.

The implementation of this method and test code are shown below:

**public** **class** PalindromeExample {

**public** **static** **void** main(String[] args) {

String s = "dened";

System.***out***.println(*isPalindrome*(s));

}

**public** **static** **boolean** isPalindrome(String s) {

**if**( s.length() <= 1 ) { // base case

**return** **true**;

}

**else** **if**( s.charAt(0) != s.charAt(s.length()-1 )) { // base case

**return** **false**;

}

**else** {

**return** *isPalindrome*(s.substring(1,s.length()-1));

}

}

}

Run this code in the visualizer:

1. Copy the code above and run in the visualizer, <http://cscircles.cemc.uwaterloo.ca/java_visualize/>
2. Notice that as the stack grows each frame contains shorter and shorter versions of the initial string.
3. Choose “Edit Code” and then select the “options” that says “Show String/Integer/etc objects, not just value”. Run the visualizer again. Objects are not stored on the stack, but the stack has pointers to them.
4. Edit the value of *s* and rerun with: “abba” (note that string length is even), and then “abcca”.

## Exercises

1. (Solution in *exercises\_primitive\_no\_helper* package, *Facorial.java*). Write a recursive method, *factorial* to compute the factorial of an input integer. Yes, we just did this. But make sure you can write it quickly as this is the standard first example of recursion. You wouldn’t want to fumble on this one.
2. (Solution in *exercises\_primitive\_no\_helper* package, *IsPalindrome.java*). Write a recursive method, *isPalindrome* that accepts a string and returns whether it is a palindrome or not. A string is a palindrome if it reads the same forwards and backwards. Yes, we just did this. But make sure you can write it quickly.
3. (Solution in *exercises\_primitive\_no\_helper* package, *SumSeries2.java*). Write a recursive method, *sumSeries2* to compute the following series:
4. (Solution in *exercises\_primitive\_no\_helper* package, *SumSeries3.java*). Write a recursive method, *SumSeries3* to compute the following series:
5. (Solution in *exercises\_primitive\_no\_helper* package, *IsAllCharacters.java*). Write a recursive method, *isAllCharacters* that accepts a string and returns *true* if all the characters in the string are letters. You will need: *Character.isLetter(c:char):boolean*. Hint: this will take some thought to get the base case correct. Write code to test!
6. (Solution in *exercises\_primitive\_no\_helper* package, *ChangePi.java*). Write a recursive method, *changePi* that accepts a string and returns a new string where all appearances of "pi" are replaced by "3.14". For example:

changePi("xpix")="x3.14x"

changePi("pipi")="3.143.14"

changePi("pip")="3.14p"

changePi("abcpi")="abc3.14"

changePi("abcpidefpighpipipi")="abc3.14def3.14gh3.143.143.14"

changePi("")=""

Hint: an idea for the recursive step:

String changePi(str)

// Need stopping rule here

// Recursive Step:

If first two characters are “pi”

Return “3.14” + changePi(first two characters removed)

Else

Return first character + changePi(first character removed)

# Examples: Returns Nothing (prints), No Helper

Some recursive methods don’t return anything. In the examples that we consider that follow, they will simply print something. In more advanced situations, a recursive method that doesn’t return anything (or even if it does), might modify an object parameter that is passed in. Remember that object parameters are passed by reference, so there is only one physical object in memory. Thus, each recursive call could make changes to that one object. This is very useful, but also can be very confusing. We won’t consider such cases.

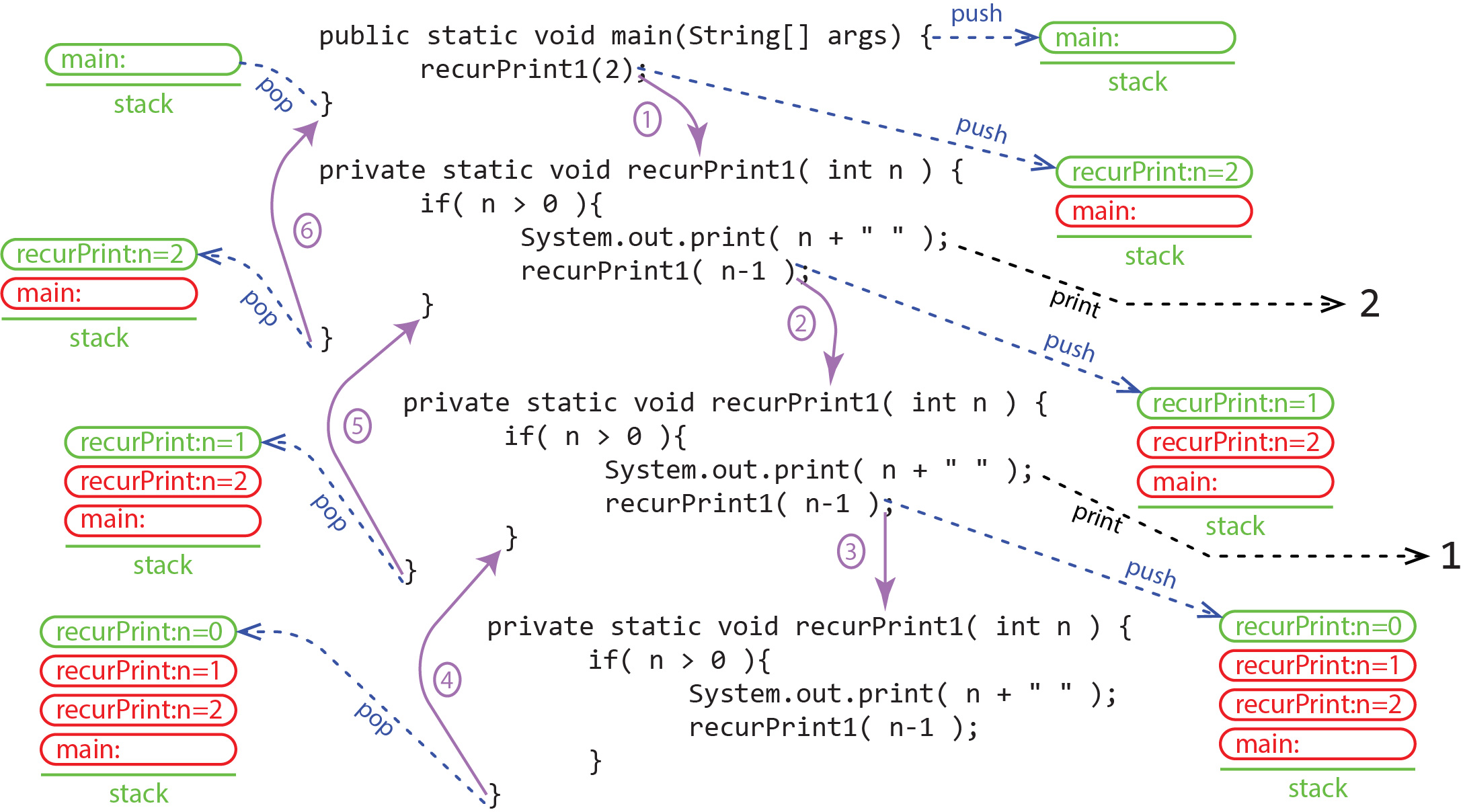
## recursivePrint

Consider the behavior of three similar methods to print integers. The situation is summarized in the table below. These show the subtleties that can exist in seemingly innocuous code. And, more specifically, when the work takes place (winding, unwinding, or both-not shown in examples). These methods are found in the *examples\_void\_no\_helper* package, *RecursivePrint.java*.

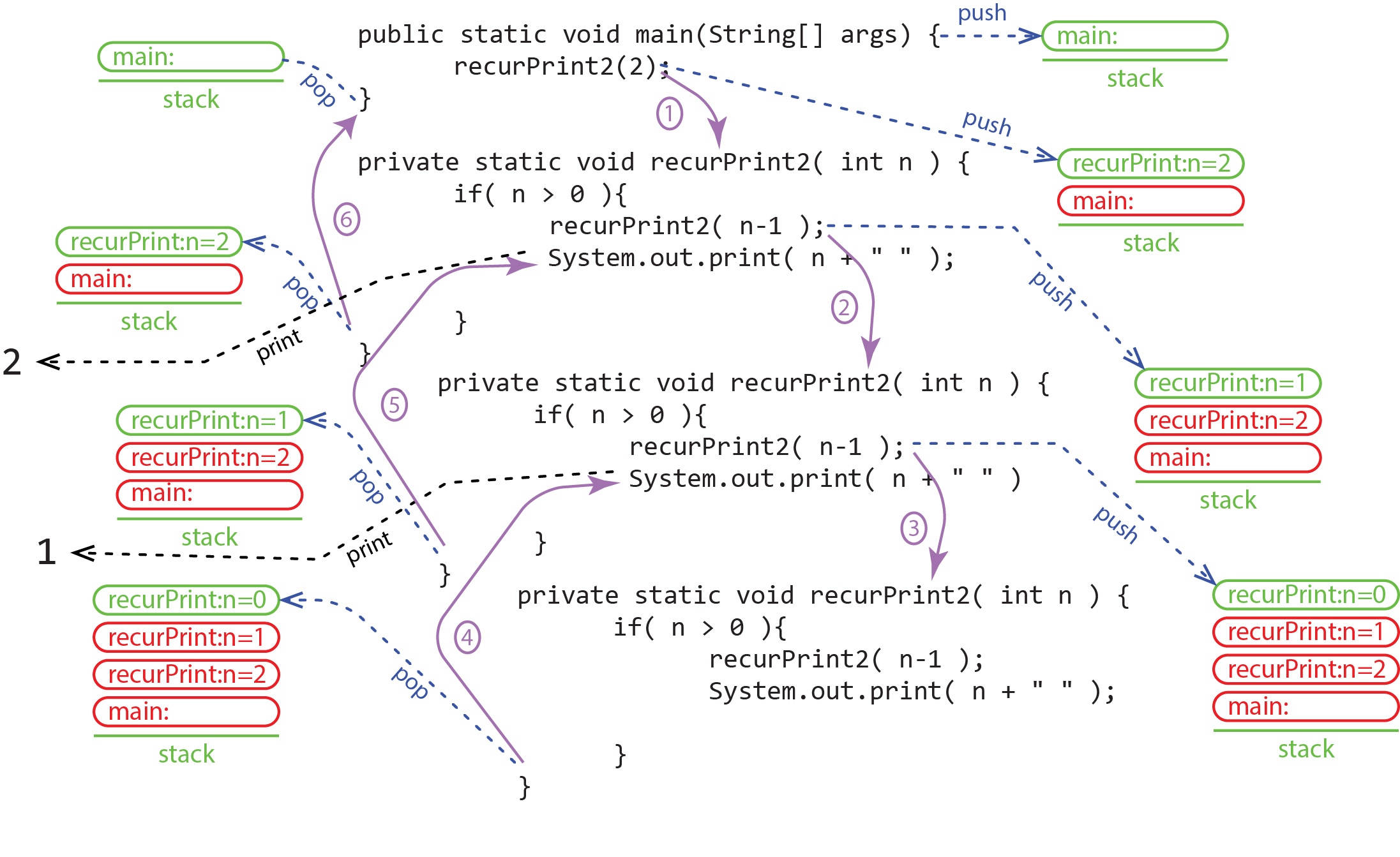
|  |  |  |
| --- | --- | --- |
| Ver | Method | Sample Call & Output |
| 1 | **private** **static** **void** recursivePrint1( **int** n ) {  **if**( n > 0 ){  System.***out***.print( n + " " );  *recursivePrint1*( n-1 );  }  } | recursivePrint1(3)  3 2 1  Prints while winding (recursing) |
| 2 | **private** **static** **void** recursivePrint2( **int** n ) {  **if**( n > 0 ){  *recursivePrint2*( n-1 );  System.***out***.print( n + " " );  }  } | recursivePrint2(3)  1 2 3  Prints while unwinding |
| 3 | **private** **static** **void** recursivePrint3( **int** n ) {  **if**( n > 0 ){  *recursivePrint3*( --n );  System.***out***.print( n + " " );  }  } | recursivePrint3(3)  0 1 2  Prints while unwinding, but changes the value of *n* before the recursive call. |

Notice also in these examples, there is no explicit base case. However, the base case is implicit – if *n* is not greater than 0, the method simply ends.

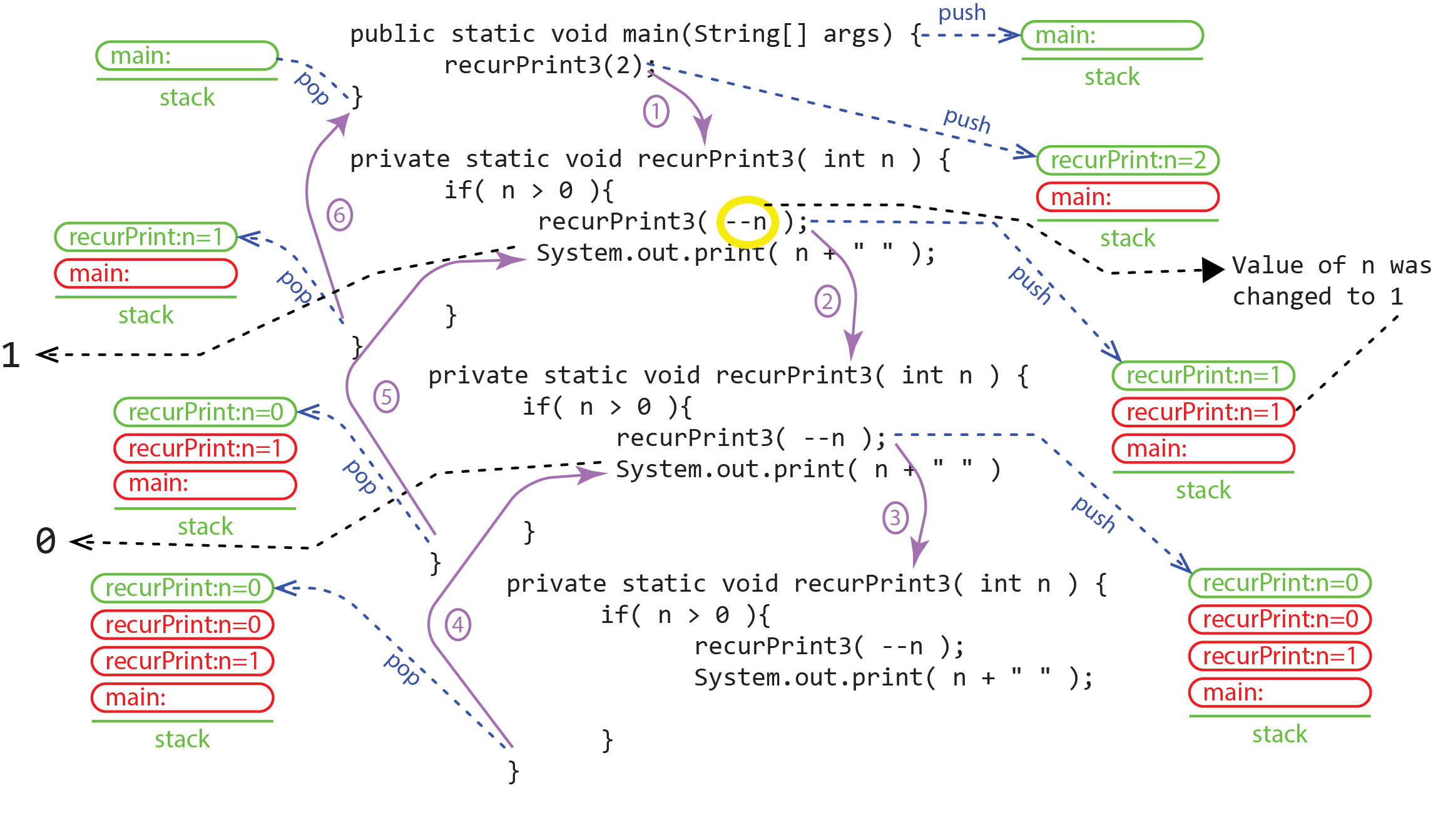
Consider the behavior of stack for *recursivePrint1*. Notice that the printing takes places during the winding.



Consider the behavior of stack for *recursivePrint2*. Notice that the printing takes places during the unwinding.



Finally, consider the behavior of stack for *recursivePrint3*. Notice that the printing takes places during the unwinding. However, more importantly, notice that the value of *n* is changed during each recursive call. Thus, when it unwinds, the value it prints the changed value of *n.*



## printString

(Solution in *examples\_void\_no\_helper* package, *PrintString*.java). Write a recursive method, *printString* that accepts a string and an integer. The method should print the string the number of times specified by the integer.

The idea for the recursive step is to print the string, the recurse decrementing the value of the integer. Thus, the solution with an explicit base case:

**public** **static** **void** printString(String msg, **int** n) {

**if**(n<=0) { // base case

**return**;

}

System.***out***.println(msg);

*printString*(msg, n-1); // recursive step

}

Or, the solution with an implicit base case:

**public** **static** **void** printString(String msg, **int** n) {

**if**(n>0) { // recursive step

System.***out***.println(msg);

*printString*(msg, n-1);

}

// implicit base case, simply return when n<=0

}

## printReverseString

(Solution in *examples\_void\_no\_helper* package, *PrintStringInReverse .java*). Write a recursive method, *printReverseString* that accepts a string and displays the string reversely on the console. For example: *printReverseString* *(“abcd”)* displays: “dcba”.

The idea for the recursive step: print last character, remove last character, recurse. We continue this process as long as the string has length of at least 1.

The implementation of the algorithm:

**private** **static** **void** printReverseString(String s) {

**if**(s.length()>0) {

System.***out***.print(s.charAt(s.length()-1));

*printReverseString*(s.substring(0,s.length()-1));

}

// implicit base case, simply returns automatically

// when length is 0.

}

## Exercises

1. (Solution in *exercises\_void\_no\_helper* package, *PrintIntegerInReverse.java*). Write a recursive method, *printReverse* that accepts an integer and displays it in reverse on the console. For example: *printReverse* (12345) displays 54321. Hint: (a) *n*%10 returns the last digit of *n.* (b) *n*/10 removes the last digit of *n.* For example: 12345%10 = 5, and 12345/10 = 1234 (c) What is the base case? When you reach a value of *n* that is less than 10.
2. (Solution in *exercises\_void\_no\_helper* package, *ReturnIntegerInReverseString.java*). Write a recursive method, *reverseString* that accepts an integer and returns a string containing the characters in the integer in reverse. In other words, same as last problem except build and return the string in reverse. Hint: whatever you “return” must have be preceded with “”, to force the integers that follow to be strings. For example: return “” + ??? + reverseString(???)
3. (Solution in *exercise\_recursivePrint* package) Consider the examples in [Section 5.1](#_recursivePrint). The three methods below are slight versions. What does each produce when called with *n=3*?

**private** **static** **void** recursivePrint4( **int** n ) {

**if**( n > 0 ) {

System.***out***.print( n + " " );

*recursivePrint4*( n-1 );

System.***out***.print( n + " " );

}

}

**private** **static** **void** recursivePrint5( **int** n ) {

**if**( n > 0 ) {

System.***out***.print( n + " " );

*recursivePrint5*( --n );

System.***out***.print( n + " " );

}

}

**private** **static** **void** recursivePrint6( **int** n ) {

**if**( n > 0 ) {

*recursivePrint6*( n-- );

System.***out***.print( n + " " );

}

}

# Examples: Recursive Helper Methods

For some problems that we want to solve recursively, we must utilize a *recursive helper method* that contains additional parameters. In other words, you are required to write a method: *recursiveMethod(n:int*) but it can’t be done (and/or efficiently) unless you introduce another parameter, *recursiveMethod (n:int, x:int)*. In this case, the first method is public and not recursive, but it calls the private *recursive helper method*. We will see what this mean as we go along.

## sumArray

(Solution in *examples\_helper\_primitive* package, *SumArray.java*). Write a recursive method, *sumArray* that accepts an array of integers and returns the sum. For example:

int[] vals = {3,6,9,8}

System.out.println( sum(vals) ) // displays 26

First, let’s write the header for the method:

**public** **static** **int** sumArray(**int**[] vals) {

An idea for the recursive step that is similar to some of the string problems we have worked is to return the first value in the array added to a recursive call with the array with the first item removed:

return vals[0] + sumArray( vals (with first item removed) )

However, we can’t “remove” a value from the array as we can remove a character from a string. We could create a new array with the first element removed, but that would not be efficient.

Some problems can’t be solved efficiently without introducing one or more parameters to the original method signature.

So, another idea is to introduce a new parameter, *loc*, which represents the location of the current element we are processing:

**private** **static** **int** sumArray(**int**[] vals, int loc) {

return vals[loc] + sumArray(vals, loc+1)

We can start at the beginning of the array (loc=0), and work towards the end. Thus, each time we recurse, *loc* is incremented, moving us closer to the base case, when *loc* equals the length of the array:

**if**(loc>=vals.length) { // base case

**return** 0;

}

To implementation this method:

**private** **static** **int** sumArray(**int**[] vals, **int** loc) {

**if**(loc>=vals.length) { // base case

**return** 0;

}

**return** vals[loc] + *sumArray*(vals, loc+1); // recursive step

}

This method is a *recursive helper method*. It is called by the public method we were originally asked to write:

**public** **static** **int** sumArray(**int**[] vals) {

**return** *sumArray*(vals,0);

}

## countCodeAbc

(Solution in *examples\_helper\_primitive* package, *CountCodeABC.java*). Write a recursive method, *countCodeAbc* that accepts a string and returns the number of occurrences of the string “abc” in the input string. The implementation must be efficient for large strings. For example:

countCodeAbc("abdcabd")=0

countCodeAbc("abcdefcdegef")=1

countCodeAbc("abcdefabcegef")=2

countCodeAbc("abcabcababc")=3

What does the last requirement, “The implementation must be efficient for large strings” mean?

* As we remember, the *String* class is immutable and thus, were the *substring* method to be used in a recursive method, it would create a new (smaller) string with each recursive call. As we recurse, each of these strings is pushed on to the stack (technically, the heap). Thus, as the stack grows, more and more strings are held in memory. This would be inefficient if the input string is very long. Thus, to make our algorithm efficient, we can’t use *substring* to recurse with a smaller string.
* We need to use a helper method so that we don’t repeatedly create new strings on the stack. So, we introduce a new parameter, *loc* to specify where we are in the string as we are processing it.

The header for the method is:

**public** **static** **int** countCodeAbc(String msg)

The header for the recursive helper method:

**private** **static** **int** countCodeAbc(String msg, **int** loc) {

The idea is to use *loc* to index through the string. At each recursive call, if the current three character are “abc” then add 1 and recurse, increasing *loc* by 3:

**if**(msg.substring(loc,loc+3).equals("abc")) {

**return** 1 + *countCodeAbc*(msg,loc+3);

}

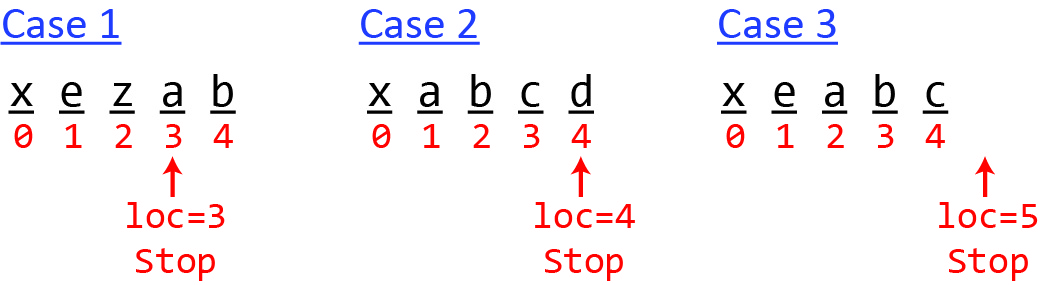
If there is not a match, then simply recurse, increasing *loc* by 1:

**else** {

**return** 0 + *countCodeAbc*(msg,loc+1);

}

What do we use for a stopping rule? We can’t go all the way to the end of the string; there must be at least 3 more characters in the string. For example, consider the 3 cases below and note that in each case the length of the string is 5. In the first case, at some point in the recursion, *loc* would end up with the value 3, which is where we should stop. In the second case, *loc* would end up with the value 4, which is where we should stop. Finally, in case 3, *loc* would end up with the value 5, which is where we should stop. if *loc* was the last character in the string, and we were looking for “abc”, we would be looking beyond the end of the string, which, of course would give a run-time error.



Thus, we should stop when *loc* is on or after the next to last character. In other words, when there are 2 or fewer characters left:

**if**(loc >= msg.length()-2) { // base case

**return** 0;

}

Another way to model the stopping rule, is to stop when the number of characters left in the string, considering the current value of *loc*, is less than or equal to 2, as shown below. Of course, using simple algebra, this result is exactly the same as the previous rule

**int** numCharsLeft = msg.length() - loc;

**if**(numCharsLeft <= 2) { // base case

**return** 0;

}

The implementation of the helper method is:

**private** **static** **int** countCodeAbc(String msg, **int** loc) {

**if**(loc >= msg.length()-2) { // base case

**return** 0;

}

**else** {

**if**(msg.substring(loc,loc+3).equals("abc"))

**return** 1 + *countCodeAbc*(msg,loc+3); // If match, jump ahead 3

**else**

**return** 0 + *countCodeAbc*(msg,loc+1); // If no match, jump ahead 1

}

}

Finally, the public method that calls the helper:

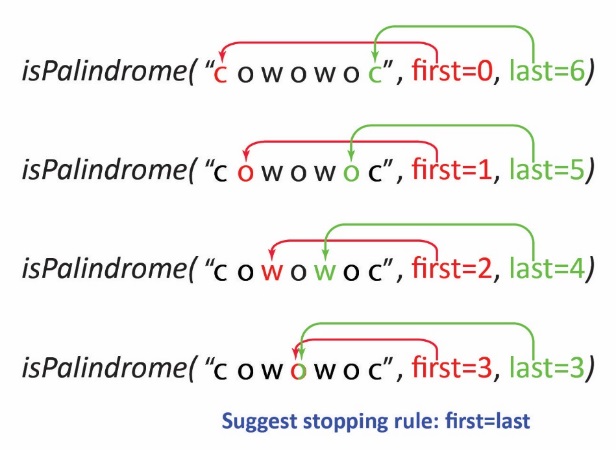
**private** **static** **int** countCodeAbc(String msg) {

**return** *countCodeAbc*(msg, 0);

}

## isPalindrone

(Solution in *examples\_helper\_primitive* package, *IsPalindrone.java*). Write a recursive method, isPalindrome that accepts a string and returns whether it is a palindrome or not. A string is a palindrome if it reads the same forwards and backwards. The implementation must be efficient for large strings

****Consider an approach where we use a recursive helper method which introduces indices to indicate which characters we are comparing during each recursive call. As shown on the right we never modify the string we are checking and we also include two additional parameters, *first* and *last* that specify which two characters we are going to compare. Initially, we consider the first (index 0) and last character (index 6). Since the characters are the same, we move *first* and *last* inward to 1 and 5, respectively. Next, since these two characters (‘o’) are the same we move *first* and *last* inward again to 2 and 4, respectively. Finally, we get to a point where *first* and *last* are point to the same character, ‘o’ which by definition is a palindrome. Thus, this suggests a stopping rule that when *first* and *last* are equal, return true (we will have to modify this shortly as it doesn’t handle the case when the input string has even length).

The structure of this approach is to define a public method that accepts a string, just as before. However, it calls a recursive helper method:

**public** **static** **boolean** isPalindrome(String s) {

**if**(s.length() <= 1 )

**return** **true**;

**return** *isPalindrome*(s,0,s.length()-1);

}

// Recursive helper method

**private** **static** **boolean** isPalindrome(String s, **int** first, **int** last) {

**if**(first==last) // base case – not correct, needs modification

**return** **true**;

**else** **if**(s.charAt(first) != s.charAt(last)) // base case

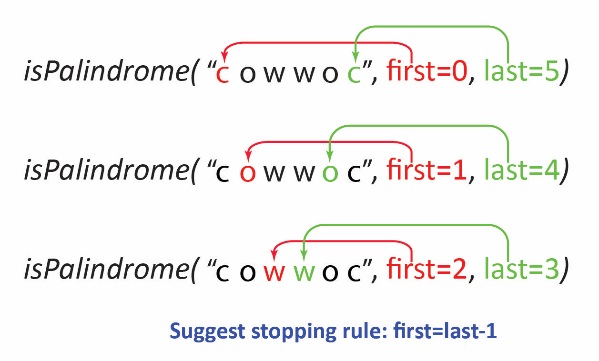
**return** **false**;

**else**

**return** *isPalindrome*(s,first+1,last-1);

}

Notice that there are two base cases: one for detecting that a string is a palindrome and one for detecting that it is not.

Our stopping rule above does not work when the input string has an even length. Consider the figure on the right where the input string has an even length. At the third iteration, we check the middle to characters (“ww”). We could develop a stopping rule that would stop there, but it would be a bit more complicated than it first appears. Every time we find that two characters are the same, before recursing, we would need to check if first=last-1 as shown below:

**private** **static** **boolean** isPalindrome2(String s, **int** first, **int** last) {

**if**(first==last) { // base case for odd length string

**return** **true**;

}

**else** **if**(s.charAt(first) != s.charAt(last)) {

**return** **false**;

}

**else** {

**if**(first==last-1) { // base case for even length string

**return** **true**;

}

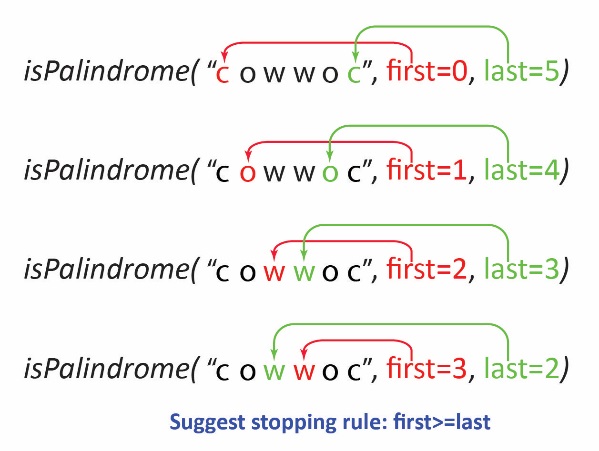
**else** {

**return** *isPalindrome2*(s,first+1,last-1);

}

}

}

However, if we recurse one more time, as shown in the figure on the right we see that first>last which clearly indicates that we are done. Now, if we combine that with the stopping rule for an odd length string we arrive at a single stopping rule: first>=last that takes care of both even and odd length strings as shown below:

**private** **static** **boolean** isPalindrome(String s, **int** first, **int** last) {

**if**(first>=last) { // base case

**return** **true**;

}

**else** **if**(s.charAt(first) != s.charAt(last)) { // base case

**return** **false**;

}

**else** {

**return** *isPalindrome*(s,first+1,last-1);

}

}

## Exercises

(Solution in *exercises\_helper\_primtive* package, *HasTimesTen.java*).

1. (Solution in *exercises\_helper\_primtive* package, *HasTimesTen.java*). Write a recursive method, *hasTimesTen* that accepts an array of integers and returns *true* if the array contains a value such that the next value in the array is that value times 10. For example: [4,8,12,15,29] is *false* and [4,8,12,120,40,400] is *true*.
2. (Solution in *exercises\_helper\_primtive* package, *IsPalindrome.java*). Write a recursive method, *isPalindrome* that accepts a string and returns whether it is a palindrome or not. To be more efficient, this method should not modify the input string. Yes, we just did this problem in the previous section. Do it again, working out the details yourself.
3. (Solution in *exercises\_helper\_primtive* package, *CountCode.java*). Write a recursive method, *countCode* that accepts a string, *msg* and a string, *code* and returns the number of non-overlapping occurrences of *code* in *msg.* To be more efficient, this method should not modify the input string, *msg*. Thus, you should use a helper method. You should not use the *contains* or *indexOf*. For example:

countCode("abcdefcd","cde")=1

countCode("bcbc","bc")=2

countCode("beeeef","ee")=2

countCode("beeeeef","ee")=2

countCode("abcdbcbcabc","bc")=4

countCode("sslsslcabssslcdresdessl","sslc")=2

# Examples: Helper, Store Intermediate Value

The examples in this section utilize a recursive helper method which introduces a parameter that represents a partial computation. In other words, it is a way to store an intermediate result. In these types of problems, the base case will return this value, which is then passed back through the unwinding.

## minInt

(Solution in *examples\_helper\_store\_intermediate* package, *MinInt.java*). Write a recursive method, *minInt* that accepts an array of integer and returns the smallest value in the array. For example:

The signature of the method is:

**public** **static** **int** minInt(**int**[] vals) {

As we have seen previously, when we are recursing on an array or list, we need to use a recursive helper method that introduces a parameter to keep track of where we are in the array/list. For example, we might take an approach like this:

**public** **static** **int** minIntIncorrect(**int**[] vals) {

**if**(vals.length > 0) {

**return** *minIntIncorrect*(vals, 0);

}

**return** Integer.***MIN\_VALUE***;

}

**private** **static** **int** minIntIncorrect(**int**[] vals, **int** pos) {

**int** minVal = vals[0];

**if**(pos >= vals.length) {

**return** minVal;

}

**if**(vals[pos] < minVal) {

minVal = vals[pos];

}

**return** *minIntIncorrect*(vals, ++pos);

}

However, this approach is incorrect. This problem is different in the sense that we are not building an answer with the recursive calls (factorial, sumSeries, countChars), we need to remember the smallest element in the array. Note:

* This method will always return the first element in the array.
* Every time *minInt* is recursively called, *minVal* is set to the first element in the array. In other words, a recursive call may update *minVal,* but as soon as the next recursive call takes place, it is reset to the first element. In other words, *minVal* is not remembered.
* **The solution is to introduce another parameter in the signature to store the current minimum length string. The rest of the algorithm is correct.**

Thus, an implementation of this algorithm:

**public** **static** **int** minInt(**int**[] vals) {

**if**(vals.length > 0) {

**return** *minInt*(vals, 0, Integer.***MAX\_VALUE***);

}

**return** Integer.***MIN\_VALUE***;

}

**private** **static** **int** minInt(**int**[] vals, **int** pos, **int** minVal ) {

**if**(pos >= vals.length) {

**return** minVal;

}

**if**(vals[pos] < minVal) {

minVal = vals[pos];

}

**return** *minInt*(vals, ++pos, minVal);

}

## minString

(Solution in *examples\_helper\_store\_intermediate* package, *MinString.java*). Write a recursive method, *minString* that accepts an *ArrayList* of strings and returns the string with the shortest length (there could be more than one, but you should return the first one encountered). For example:

minString([abc, ab, ddefg, zz, kade])=ab

minString([a])=a

This problem is very similar to the previous one in that we need a recursive helper method that introduces a location parameter and a parameter to store the intermediate result. Thus, the recursive helper method:

**private** **static** String minString(ArrayList<String> strs, **int** loc, String minStr) {

**if**(loc==strs.size()) { // base case

**return** minStr;

}

// Recursive step. First, check to see if

// cur string is shorter than cur min

**if**(strs.get(loc).length()<minStr.length()) {

minStr = strs.get(loc);

}

**return** *minString*(strs, loc+1, minStr);

}

The public method:

**public** **static** String minString(ArrayList<String> strs) {

**if**(strs.size()==0 || strs==**null**) {

**return** **null**;

}

**return** *minString*(strs, 0, strs.get(0));

}

Finally, a sample call to the method:

ArrayList<String> vals = **new** ArrayList<>(

Arrays.*asList*("abc", "ab", "ddefg", "zz", "kade"));

String minLenStr = *minString*(vals);

## factorial

(Solution in *examples\_helper\_store\_intermediate* package, *Factorial.java*). We will use this technique of storing the intermediate value/result as a parameter in a recursive helper method with the factorial problem. In the version of factorial, we considered [earlier](#_Example:_Factorial), we built the result in the return statement:

**public** **static** **long** factorial(**long** n) {

**if**(n==1 || n==0) {

**return** 1;

}

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

}

Instead, we will introduce a parameter in a helper method to store the intermediate result. When we hit the base case, we simply return the result. Otherwise, we compute the intermediate result and then pass it to the recursive call.

**private** **static** **long** factorial(**long** n, **int** result ) {

**if**(n<=1) {

**return** result;

}

result \*= n;

**return** *factorial*(n-1, result);

}

Then the calling method simply initializes the call to the recursive method with the value of 1.

**public** **static** **long** factorial(**long** n) {

**if**(n>=0) {

**return** *factorial*(n, 1); // recursive step

}

**return** Long.***MIN\_VALUE***;

}

## Exercises

1. (Solution in *exercises\_helper\_store\_intermediate* package, *ShortStrings.java*). Write a recursive method, *shortStrings* that accepts an ArrayList of strings and an integer, *len.* It should return an ArrayList of strings whose length is less than *len*.
2. (Solution in *exercises\_helper\_store\_intermediate* package, *Power.java*).Write a recursive method, *pow(x,n)* that raises *x* to the power of *n, e.g.* . The solution should employ a helper method that stores the intermediate result as a parameter.

# Binary Search

A common task in software systems is to search for an item in a list (collection). The item we are searching for is called the *key*, which may or may not be in the list. If we find an item in the list that matches the key, then we either return the item, or we return the index of the item (if the list is an index-based structure).

If the items in the list are unordered, then there is not much we can do except inspect the items one-by-one looking for the *key.* This is called a *brute-force search* or *linear search*. If a list has *n* items, in the best case, we find it on the first inspection. In the worst case, we look through all items and don’t find it, or find it in the last position. On average, if the item exists in the list, we find it after inspections.

If the items in the list are ordered, then the *fastest* way to search for a key is to use the *binary search* algorithm.We show how binary search works with several examples. It is customary for the binary search algorithm to return the index where the key was found. If the key is not found, a negative integer is returned, which has a special meaning we discuss shortly.

## Example 1

Suppose we are looking for 11 in the array below:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| BinarySearch | ( | vals= | |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | | 2 | 4 | 7 | 10 | 11 | 45 | 50 | 59 | 60 | 66 | 69 | 70 | 79 | | , key=11 | ) |

**Iteration 1:**

1. Find index of the middle of the list, (0+12)/2=6. Then, since key<vals[6], call BinarySearch on the left half of the list:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| BinarySearch | ( | vals= | |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | | 2 | 4 | 7 | 10 | 11 | 45 | 50 | 59 | 60 | 66 | 69 | 70 | 79 | | , key=11 | ) |

1. Then, since key<vals[6], call BinarySearch on the left half of the list:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| BinarySearch | ( | vals= | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | 5 | | 2 | 4 | 7 | 10 | 11 | 45 | | , key=11 | ) |

**Iteration 2:**

1. Find index of the middle of the list, (0+5)/2=2.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| BinarySearch | ( | vals= | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | 5 | | 2 | 4 | 7 | 10 | 11 | 45 | | , key=11 | ) |

1. Then, since key>vals[2] , call BinarySearch on the right half of the current list:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| BinarySearch | ( | vals= | |  |  |  | | --- | --- | --- | | 3 | 4 | 5 | | 10 | 11 | 45 | | , key=11 | ) |

**Iteration 3:**

1. Find index of the middle of the list, (3+5)/2=4.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| BinarySearch | ( | vals= | |  |  |  | | --- | --- | --- | | 3 | 4 | 5 | | 10 | 11 | 45 | | , key=11 | ) |

1. Then, since key==vals[4], return 4.

## Example 2

Next, we will be a bit more formal. We will introduce variables, *low* and *high* to keep track of the beginning and end of the list, respectively. Initially, *low=0* and *high=vals.length-1.* As we recurse with either the left or right sublist, we will adjust *low* and *high* appropriately to represent the range of indices that represent the sublist.

Suppose we are looking for 62 in the array below. This example shows what happens when the *key* is not found and will help us develop a base case

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| BinarySearch | ( | vals= | |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | | 2 | 4 | 7 | 10 | 11 | 45 | 50 | 59 | 60 | 66 | 69 | 70 | 79 | | , key=62 | ) |

**Iteration 1:**

1. Let’s introduce variables, *low* and *high* to keep track of the beginning and end of the list, respectively:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| low  ↓ |  |  |  |  |  |  |  |  |  |  |  | high  ↓ |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| 2 | 4 | 7 | 10 | 11 | 45 | 50 | 59 | 60 | 66 | 69 | 70 | 79 |

1. As before, find index of middle value: .

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| low  ↓ |  |  |  |  |  | mid  ↓ |  |  |  |  |  | high  ↓ |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| 2 | 4 | 7 | 10 | 11 | 45 | 50 | 59 | 60 | 66 | 69 | 70 | 79 |

1. Then, since: key>vals[6], change and call BinarySearch on the list indexed from *low* to *high.*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| BinarySearch | ( | vals= | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | low  ↓ |  |  |  |  | high  ↓ | | 7 | 8 | 9 | 10 | 11 | 12 | | 59 | 60 | 66 | 69 | 70 | 79 | | , key=62 | ) |

**Iteration 2:**

1. Find index of middle value: .

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| BinarySearch | ( | vals= | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | low  ↓ |  | mid  ↓ |  |  | high  ↓ | | 7 | 8 | 9 | 10 | 11 | 12 | | 59 | 60 | 66 | 69 | 70 | 79 | | , key=62 | ) |

1. Then, since key<vals[9] , change , and call BinarySearch on the list indexed from *low* to *high.*

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| BinarySearch | ( | vals= | |  |  | | --- | --- | | low  ↓ | high  ↓ | | 7 | 8 | | 59 | 60 | | , key=62 | ) |

**Iteration 3:**

1. Find index of middle value: .

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| BinarySearch | ( | vals= | |  |  | | --- | --- | | low, mid  ↓ | high  ↓ | | 7 | 8 | | 59 | 60 | | , key=62 | ) |

1. Then, since key>vals[7] , change , and call BinarySearch on the list indexed from low to high.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| BinarySearch | ( | vals= | |  | | --- | | low, high  ↓ | | 8 | | 60 | | , key=62 | ) |

**Iteration 4:**

1. Find index of middle value: .

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| BinarySearch | ( | vals= | |  | | --- | | low, mid, high  ↓ | | 8 | | 60 | | , key=62 | ) |

Since low, mid, and high are all the same this indicates that the key is not present. However, when the key is not found, it is customary to return the location where the key would occur if it were inserted into the list (sort of, we will clarify this shortly). One way to do this is to continue the algorithm:

1. Then, since key>vals[8] , change , and call BinarySearch on the list indexed from *low* to *high.*

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| BinarySearch | ( | vals= | |  |  | | --- | --- | | high  ↓ | low  ↓ | | 8 | 9 | | 60 | 66 | | , key=62 | ) |

**Iteration 5:**

1. Now, since , this is the base case: it means that the was not found, and, that it falls between and . Thus, if the key were to be inserted in the proper order, it would be at index 9 (the 10th element). It is customary for the binary search algorithm to return -1. For this example, -10. The negative sign means that the key was not found. This begs the question as to why the algorithm doesn’t just return -9 since 9 is the index where it belongs. See the next two examples!

## Example 3

Suppose we are looking for 2 in the array below.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| BinarySearch | ( | vals= | |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | | 2 | 4 | 7 | 10 | 11 | 45 | 50 | 59 | 60 | 66 | 69 | 70 | 79 | | , key=2 | ) |

The method returns 0 because that is the index where the key was found, vals[0].

Next, suppose we are looking for 1 in the array (same as previous array) below.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| BinarySearch | ( | vals= | |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | | 2 | 4 | 7 | 10 | 11 | 45 | 50 | 59 | 60 | 66 | 69 | 70 | 79 | | , key=1 | ) |

Of course, *binarySearch* will not find the key. Now, suppose we’ve coded it so that when it failed to find the *key* it returned the negative of the index of where it would belong in the list if it were inserted. Then, in the example above, “1” belongs at index 0, so we return -0, which is just 0. Thus, we see that the results would be ambiguous when 0 is returned. Does 0 mean the key was found or does it mean the key belongs in the element at index=0? Thus, the standard version of *binarySearch* returns the negative of the position in the list, *e.g.* return –low-1. For the example above, *binarySearch* returns -1. We say a bit more about this at the end of this section.

## Development of the Binary Search Algorithm

Let’s summarize with a rough draft of a recursive algorithm to do binary search.

BinarySearch( list, key )

Find middle item

If key=middle

Return index of middle

Else if key < middle

BinarySearch( leftSubList, key )

Else

BinarySearch( rightSubList, key )

We have two issues to resolve:

1. How do we tell the algorithm to search on the left (or right) sublist?
2. How do we define a stopping rule?

We’ll need to think carefully about this. Let’s assume we have an index-based list that is numbered from 0 to *length-1*. As we saw earlier, let’s indicate the portion of the list that we want to consider with the variables and . Then, we can find the index of the middle item with:

And thus, the middle item is: and:

* is the elements from to
* is the elements from to

Now, how do we know when to stop the recursion? As we said earlier:

if low > high // *key* not found

return –low-1

## Binary Search implementation for an array of integers.

Solution in *example\_binary\_search* package.

**public** **static** **int** binarySearch( **int**[] vals, **int** key ) {

**return** *binarySearch*( vals, key, 0, vals.length-1 );

}

**private** **static** **int** binarySearch( **int**[] vals, **int** key, **int** low, **int** high ) {

**if**( low > high ) // key not found

**return** -(low+1);

**else** {

**int** mid = (low+high)/2;

**if**( key < vals[mid] ) // search left sub-list

**return** *binarySearch*( vals, key, low, mid-1 );

**else** **if**( key > vals[mid] ) // search right sub-list

**return** *binarySearch*( vals, key, mid+1, high );

**else**

**return** mid;

}

}

## Final Comments on Binary Search

Binary search is incredibly fast. Suppose we have items in a list. It turns out that the maximum number of iterations is . Compare that to *brute force search* which on average takes iterations. As shown in the table below, the difference is astounding!

|  |  |  |
| --- | --- | --- |
|  | **Iterations** | |
| **N** | **Binary Search – Worst Case** | **Brute Force Search – Average Case** |
| 10 | 5 | 5 |
| 100 | 8 | 50 |
| 1,000 | 11 | 500 |
| 100,000 | 18 | 50,000 |
| 1,000,000 | 21 | 500,000 |
| 1,000,000,000 | 31 | 500,000,000 |
| 1,000,000,000,000 | 41 | 500,000,000,000 |

We stated above that when binary search returns a negative number this means the item was not found. Specifically, we wrote in the algorithm above that if the item was not found, we return:

**return -(low+1);**

What does this number mean? We can use it to tell us the index of where the key would belong in the list. For example (Solution in *example\_binary\_search* package):

**int** index = *binarySearch*( vals, key );

System.***out***.println( "\nkey=" + key + ", index=" + index );

**if**( index < 0 ) { // key not found

**int** loc = -1\*index - 1; // find where key belongs

System.***out***.println( "Key belongs at index:" + loc );

**int** max; // find next value greater than key

**if**(loc < vals.length) { max = vals[loc]; }

**else** { max = Integer.***MAX\_VALUE***; }

System.***out***.println( "Next value greater than key: " + max );

}

Thus, if the return from *binarySearch* is negative, then make it positive and subtract 1. This new value tells us where we would insert this value should we want to. It also tells you where the values are that are bigger (or smaller) than the key. In other words, it provides information about where the key belongs in the list. If the modified index is equal to the length of the list, then this means that the key is larger than all items in the list. Consider this list:

**list=[1, 4, 8, 10, 17, 33]**

Now, suppose we search for *Key* as shown in the table below. i*ndex* shows the return from binary search, and *loc* shows where the belongs in the list.

|  |  |  |  |
| --- | --- | --- | --- |
| **Key** | ***index* (returned from binary search)** | ***loc=-index-1*** | **Next Item Above** |
| -4 | -1 | -(-1)-1=0 | List[0]=1 |
| 3 | -2 | -(-2)-1=1 | List[1]=4 |
| 5 | -3 | 2 | List[2]=8 |
| 9 | -4 | 3 | List[3]=10 |
| 13 | -5 | 4 | List[4]=17 |
| 28 | -6 | 5 | List[5]=33 |
| 99 | -7 | 6 | Bigger than all items in list |

Appendix

1. Video Lecture on Recursion from MIT

A lecture on recursion from MIT: <https://web.mit.edu/6.005/www/fa15/classes/10-recursion/>

1. Factorial Example

In mathematics, the factorial of a number, is the product of all positive integers from 1 to *.* For example:

|  |  |
| --- | --- |
| Examples | Alternate (Recursive Representation) |
|  |  |

Also, by definition. Thus, the *recursive* *definition* of factorial is:

where for

For example, we can see the recursive nature when we look at an example. As shown below, to calculate 4!, we break it into 4\*3!. At the next iteration (a recursive call), we see that 3! is broken into 3\*2!, *etc.*

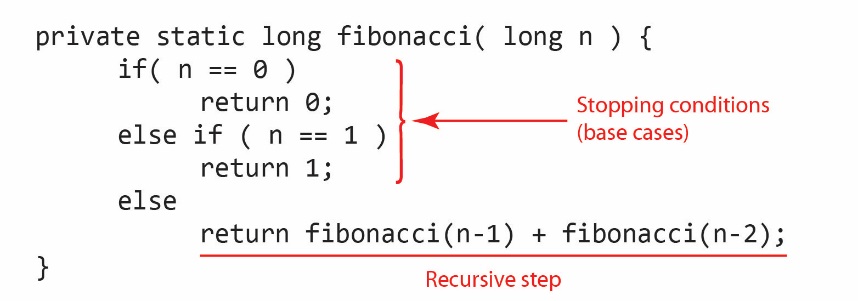
1. Problem: Computing Fibonacci Numbers

A Fibonacci number can be recursively defined as:

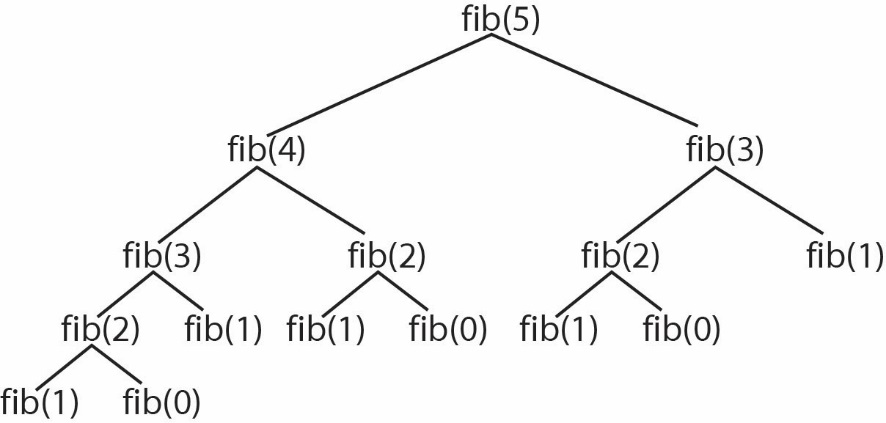
, where .

Examples:

(Solution in *examples\_primitive\_no\_helper* package, *Fibonacci.java*). Below we show a method to calculate the Fibonacci number for a given input using a recursive approach. Note that there are two base cases:



Using recursion to calculate a Fibonacci number is not efficient. We are repeatedly calling *Fibonacci* on the same values. Consider, a trace of the recursive calculation of *fibonacci(5)*



Note that we have called *fib(3)* twice, *fib(2)* 3 times, and *fib(1)* 5 times, *etc*.

* We should never use recursion when we duplicate work by solving the same instance of a problem in separate recursive calls.

Fun Fact: For it can be proved that the number of calls to is larger than the actual Finonacci number itself! For instance, when , and the total number of recursive calls is more than 300,000,000. This is particularly ridiculous considering an iterative approach would take just 40 iterations of a loop! We say this recursive algorithm has an *exponential* number of recursive calls. In other words, the number of recursive calls grows exponentially as *n* increases. The graph on the right shows the time it took to compute Fibonacci(n) on my computer using a recursive algorithm. The time for an iterative approach is effectively 0.

1. Selection Sort
2. Sorting means to put the items in a list in order where *order* is determined by implementing *Comparable*, *Comparator*, or relying on the natural ordering for primitive types. Of course we can sort a list using *Collections.sort* (or *Arrays.sort*)*;* however, in this section we study how a particular sort method, *selection sort* works. Sorting is one of the most studied algorithms in computer science:

* Sorting Algorithms: <https://en.wikipedia.org/wiki/Sorting_algorithm>
* Visual Animations: <https://www.toptal.com/developers/sorting-algorithms/>
* Auditory Animations: <https://www.youtube.com/watch?v=t8g-iYGHpEA>
* Hungarian Folk Dance – Quick Sort: <https://www.youtube.com/watch?v=ywWBy6J5gz8>
* Pancake Sorting: <https://en.wikipedia.org/wiki/Pancake_sorting>
* In Java, Arrays.sort uses Dual-Pivot [Quicksort](https://en.wikipedia.org/wiki/Quicksort) for primitives and [Timsort](https://en.wikipedia.org/wiki/Timsort) for objects. Collections.sort uses Timsort.

1. Problem statement: Assume we have an array of *n* items, indexed from 0 to n-1. We want to *sort* these items without creating another array. Although we use an array here, the idea is identical if we had used an *ArrayList* instead.
2. There are many sorting algorithms[[1]](#footnote-1). Here, we consider *Selection Sort[[2]](#footnote-2)*. An example of how it works is shown below. Suppose we start with an array of 7 integers:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 2 | 9 | 5 | 4 | 8 | 1 | 6 |

1. Find largest value in positions 0-6 (the value 9 at position 1):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 2 | 9 | 5 | 4 | 8 | 1 | 6 |

1. Swap this largest value with the value in position 6:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  | sorted |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 2 | 6 | 5 | 4 | 8 | 1 | 9 |

1. Next, find largest value in positions 0-5 (the value 8 at position 4):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  | sorted |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 2 | 6 | 5 | 4 | 8 | 1 | 9 |

1. Swap this largest value with the value in position 5:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | sorted | sorted |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 2 | 6 | 5 | 4 | 1 | 8 | 9 |

1. Continue this process …
2. Find largest value in positions 0-1 (the value 2 at position 0):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | sorted | sorted | sorted | sorted | sorted |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 2 | 1 | 4 | 5 | 6 | 8 | 9 |

1. Swap this largest value with the value in position 1. Now the list is sorted.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | sorted | sorted | sorted | sorted | sorted | sorted |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 2 | 4 | 5 | 6 | 8 | 9 |

1. Let’s think about implementing Selection Sort recursively: Here are several key observations:
2. The process of *finding the largest* is the same for a list of *n* items as it is for a list of *n-1* items
3. Think about partitioning the list into values that are sorted (the right hand portion of the list) and unsorted (the left hand portion of the list).
4. When we find the largest value and put it in the last position, then the right portion of the list consisting of the last element only is sorted. Thus, when we find the next largest and put it in the next to last position, then the right portion of the list consisting of the last two elements is sorted and no longer needs to be considered. *Etc*.
5. When there is only 1 item left in the left hand portion of the list we are done.
6. We see from above that we are going to need to keep track of the current starting position of the right hand (sorted) portion of the list. This suggests that we will need a helper method. A recursive helper algorithm might work like this:

Sort( *list*, *endPos* )

if one item left in list

return

else

Find largest in list from 0 to endPos

Swap largest item with item in endPos

Sort( list, endPos-1)

And then we could call this helper with:

Sort( list )

Sort( list, list.length-1 );

1. We will implement this algorithm assuming the input is an array of integers. (Solution in *example\_select\_sort\_integers* package)

**private** **static** **void** selectSort( **int**[] vals ) {

**if**( vals.length > 1 )

*selectSort*( vals, vals.length-1 );

}

**private** **static** **void** selectSort( **int**[] vals, **int** endPos ) {

**int** maxValue = vals[0];

**int** maxIndex = 0;

// Base case: list is sorted when only 1 element left.

**if**(endPos<1)

**return**;

// Find max

**for**( **int** i=1; i<=endPos; i++ )

**if**( vals[i] > maxValue ) {

maxValue = vals[i];

maxIndex = i;

}

// Swap max with value in endPos.

**int** lastVal = vals[endPos];

vals[endPos] = maxValue;

vals[maxIndex] = lastVal;

// Recurse with a shorter list

*selectSort*( vals, endPos-1 );

}

|  |
| --- |
| Exercise   1. (Solution in *exercise\_select\_sort\_person* package). Suppose you have a *Person* class that implements *Comparable.* Right now, it implements *Comparable* to compare *Person* objects based on their SSN, but in the future this could change. For instance, *compareTo* may be changed so that it compares based on *name*. Write *selectSort* so that it sorts an *ArrayList* of *Person* objects no matter how the *compareTo* is implemented. Hint: when finding the max, use the *compareTo* method. |

1. Tail Recursion
2. A recursive method is said to be *tail recursive* if there are no pending operations to be performed on return from a recursive call. Tail recursion is desirable because some compilers can optimize tail recursion to reduce stack space.
3. Example – This method is not tail recursive because when the recursive call, *factorial(n-1)* returns, the result must be multiplied by *n.*

**public** **static** **long** factorial( **long** n ) {

**if**( n == 0 )

**return** 1;

**else**

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

}

1. Example – This method is tail recursive:

**private** **static** **boolean** isPalindrone( String msg, **int** low, **int** high ) {

**if**( low >= high )

**return** **true**;

**if**( msg.charAt(low) == msg.charAt(high) )

**return** *isPalindrone*( msg, low+1, high-1 );

**else**

**return** **false**;

}

1. You can often convert a recursive method to a tail-recursive method by using a helper method with an extra parameter. This parameter is used to store the pending operation so that there is no longer a pending operation.
2. Example – Factorial, tail-recursive.

**public** **static** **long** factorial(**int** n) {

**return** *factorial*( n, 1 );

}

**private** **static** **long** factorial(**int** n, **long** result ) {

**if** (n == 1)

**return** result;

**else**

**return** *factorial*(n - 1, n\*result );

}

1. Example

* Not tail recursive:

**public** **static** **double** sumBack( **int** n ) {

**if**( n == 1 ) **return** 1;

**return** 1.0/n + *sumBack*(n-1);

}

* Tail recursive:

**public** **static** **double** sumBack2( **int** n ) {

**return** *sumBack2*(n,1);

}

**private** **static** **double** sumBack2( **int** n, **double** result ) {

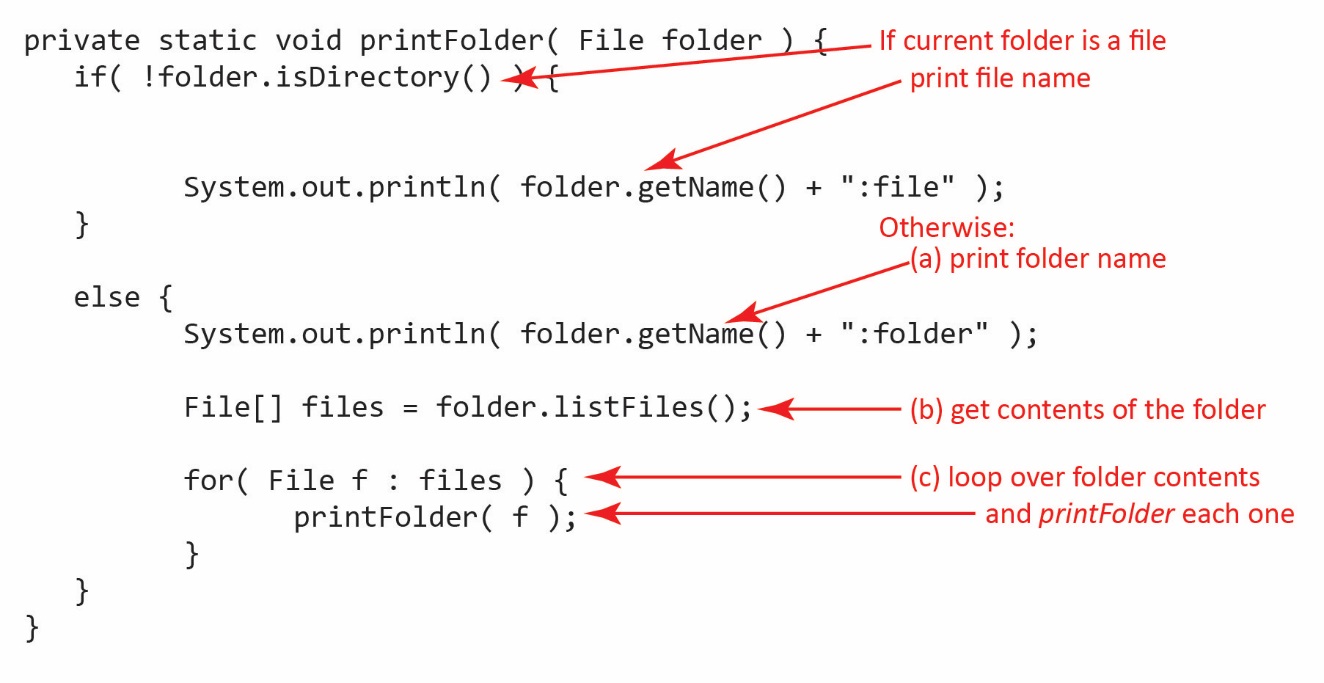
**if**( n == 1 ) **return** result;

**return** *sumBack2*(n-1, 1.0/n+result);

}

1. Recursion vs. Iteration
2. Recursion is another form of program control. It is repetition, but without a loop.
3. Recursion is more expensive than iteration in terms of time and space. Each recursive call means the JVM must create space on the stack for the local variables and parameters and put them there. This requires extra time to manage this space. However, *tail recursion* (see next section) is nearly identical to iteration.
4. Which should you use? Recursion or iteration? Use the one that provides an intuitive solution that mirrors the problem. If an iterative solution is obvious, use it as it will generally be more efficient than the recursive solution.
5. Any problem that can be solved recursively can be solved iteratively.
6. Problem: printFolder

(Solution in *example\_print\_folder* package*, ListFiles.java*, *printFolder* method). Consider a method that accepts a folder (*File* object) and prints the file names and folder names in that folder, recursively.



Notice that the base case is a bit fuzzy. The base case is the *if* block: every time you find a file, you have hit a base case and you return. Otherwise, you recurse into the folder.

|  |
| --- |
| If you want to try this yourself, copy the code:  **private** **static** **void** printFolder( File folder ) {  **if**( !folder.isDirectory() ) {  System.***out***.println( folder.getName() + ":file" );  }  **else** {  System.***out***.println( folder.getName() + ":folder" );    File[] files = folder.listFiles();    **for**( File f : files ) {  *printFolder*( f );  }  }  } |

1. Problem: printFolder with Indent

(Solution in *example\_print\_folder* package*, ListFiles.java*, *printFolderWithIndent* method). We will consider a slightly different problem than in an appendix above. Here, we consider displaying all the files and folders within a folder, recursively and having the output be indented at each level of folder.

|  |  |
| --- | --- |
| Consider folder *abc* below:  E:\Data-Classes\_CS 3340, Spring 09\WebApps\New2Draco\wp02\gview\pics\fol.tif | We would like the display to look something like this:  **abc:folder**  **b:folder**  **dd.class:file**  **dd.java:file**  **g:folder**  **ee.class:file**  **ee.java:file**  **r:folder**  **jj.docx:file**  **nn.docx:file**  **a.doc:file**  **dd2.abc:file**  **dd3.abc:file**  **dd.abc:file** |

1. Algorithm

Input: a reference to a folder.

Steps:

1. Print folder name
2. Obtain the a list of the files and folders inside the input folder.
3. Loop through the list

3a. If item is a file, print the name

3b. Else if item is a folder, print the name and go to step 1

Pseudo-code:

**PrintFolder( *inFolder* )**

**Print folder name**

***list =* Get list of files and folders contained in *inFolder***

**For each *item* in *list***

**If *item* is a file, print file name**

**Else if *item* is a folder, call PrintFolder( *item* )**

1. Implementation

|  |  |
| --- | --- |
| **import java.io.\*; public class ListFiles {  public static void main(String[] args)   {  File folder = new File("abc");  printFolder( folder );   }   private static void printFolder( File folder)  {  System.out.println( folder.getName() + ":folder");  File[] files = folder.listFiles();    for( File f : files )  {  if( !f.isDirectory() )  System.out.println( f.getName() + ":file" );  else  printFolder( f );  }  } }** | Output:  **abc:folder**  **a.doc:file**  **b:folder**  **cc.b:file**  **cc2.b:file**  **dd.abc:file**  **dd2.abc:file**  **dd3.abc:file**  **g:folder**  **a.txt:file**  **b.txt:file**  **r:folder**  **jj.doc:file**  **nn.doc:file** |

1. What could we do to make the output indent for folders? We could use a recursive helper method. Every time we recursively call the method, we increment a string that specifies how much indention.

**private static void printFolder( File folder ) {  
 printFolderWithIndent( folder, "" );  
 }  
  
 private static void printFolderWithIndent( File f, String indent ) {  
 System.out.println( indent + f.getName() + ":folder" );  
 indent += " ";  
 File[] files = f.listFiles();  
   
 for( File f2 : files ) {  
 if( !f2.isDirectory() )  
 System.out.println( indent + f2.getName() + ":file" );  
 else  
 printFolderWithIndent( f2, indent );  
 }  
 }**

Output:

**abc:folder**

**a.doc:file**

**b:folder**

**cc.b:file**

**cc2.b:file**

**dd.abc:file**

**...**

1. Problem: getFiles

(Solution in *example\_print\_folder* package*, ListFiles.java*, *getFiles* method). Consider a problem related to *printFolder* in the appendix above: write a method that accepts a folder name and returns a list of only *File* objects that are files (not folders), recursively.

Let’s call this new method *getFiles* and we should see that it is going to return a list of *File* objects. Thus, the signature will look like this:

**public** **static** ArrayList<File> getFiles( File folder ) {

We will use the *printFolder* method above as the basis to construct this new algorithm. Consider the (incorrect) solution below.

**private** **static** ArrayList<File> incorrectGetFiles( File folder ) {

ArrayList<File> files = **new** ArrayList<>();

**if**( !folder.isDirectory() ) {

files.add(folder);

**return** files;

}

**else** {

File[] localFiles = folder.listFiles();

**for**( File f : localFiles ) {

*incorrectGetFiles*( f );

}

**return** files;

}

}

The problem with this approach is that we are creating a new *ArrayList* every time we make a recursive call. For instance, when the *if* statement is *true* (*i.e.* you have a file, not a folder) you add this file to the list and return. However, you return to the *else* block of the calling method, which has an entirely different (empty!) list. This method will always return an empty list!

The problem would be easy if there was one list that was created by a *getFiles* method and passed to a *getFiles* helper method. Thus, the public *getFiles* would look like ths:

**public** **static** ArrayList<File> getFiles( File folder ) {

ArrayList<File> files = **new** ArrayList<>();

*getFiles*( folder, files );

**return** files;

}

Since *files* is a reference variable we can add *File* objects to it in the helper method. Thus, when the public method (above) is complete, *files* will contain all the *File* objects (that are files). The helper method looks like this:

**private** **static** **void** getFiles( File folder, ArrayList<File> files ) {

**if**( !folder.isDirectory() ) {

files.add(folder);

}

**else** {

File[] localFiles = folder.listFiles();

**for**( File f : localFiles ) {

*getFiles*( f, files );

}

}

}

Notice that (a) we no longer need any *return* statements, (b) the method is *private,* it is a helper method.

1. <https://en.wikipedia.org/wiki/Sorting_algorithm> [↑](#footnote-ref-1)
2. <https://en.wikipedia.org/wiki/Selection_sort> [↑](#footnote-ref-2)