# Introduction to Recursion

Davide Fossati

10 April 2015

## 1 Powers of two

Think about this question: what is the value of 2 to the power of 17? You have 5 seconds, and no calculator:

$$2^{17} = ?$$

Time's up? Well, the answer is simple, even though it may look like cheating:

$$2^{17} = 2 * 2^{16}$$

Now I'm sure you can answer these questions: what is 2 to the power of 16? And 2 to the power of 15? You got the pattern:

$$\begin{aligned} 2^{17} &= 2 * 2^{16} \\ 2^{16} &= 2 * 2^{15} \\ 2^{15} &= 2 * 2^{14} \\ &\vdots \end{aligned}$$

Eventually the "cheating" has to stop. When?

$$2^0 = 1$$

This line of reasoning leads us to a *recursive* definition of power of two. A recursive definition is composed of a *base case*, which is a simple case for which an answer can be provided directly, and a *recursive case* that builds the answer for a generic case using *the same formula*, but with arguments that are closer to the base case.

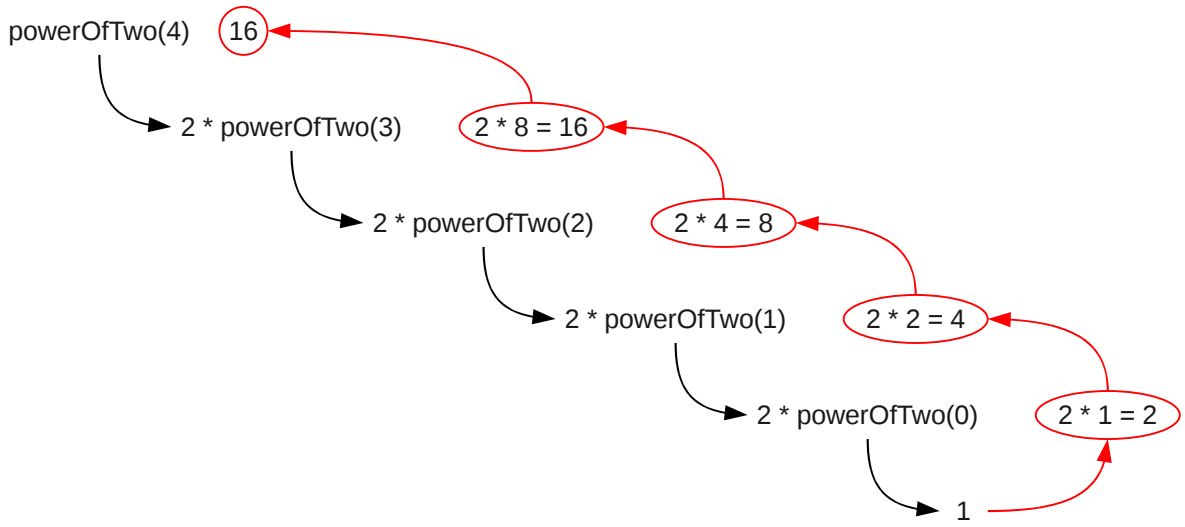
$$\begin{aligned} 2^0 &= 1 && \textbf{base case} \\ 2^N &= 2 * 2^{N-1} && \textbf{recursive case} \end{aligned}$$

We can easily code the recursive definition of power of two in Java. Java allows us to define *recursive functions*. A recursive function includes one or more calls to itself within the body of the function.

```
// Calculates 2^n using recursion
public static int powerOfTwo(int n) {
    // base case: 2^0 = 1
    if (n == 0) {
        return 1;
    } else {
        // recursive case: 2^n = 2 * 2^(n-1)
        return 2 * powerOfTwo(n-1);
    }
}
```

Notice how the function `powerOfTwo` calls itself with a value of $n$ that is closer to the base case. This is one of the secrets of recursion: the recursive call should be asked to solve a problem that is "smaller" than the original one, where "smaller" means *closer to the base case*. Since our base case is $0$, and $n$ is supposed to be greater than $0$, then $n - 1$ will be closer to the base case.

Does this function really work? Yes, it does. To convince ourselves, let's trace what happens when we call `powerOfTwo(4)`, which should return $16$ as a result.




The function `powerOfTwo(4)` enters the recursive case (because $4 > 0$), so it calls `powerOfTwo(3)` and waits for its result. In turn, `powerOfTwo(3)` calls `powerOfTwo(2)`, which calls `powerOfTwo(1)`, which calls `powerOfTwo(0)`. Finally `powerOfTwo(0)` returns a result directly without calling anyone else because it is the base case. The value 1 is now returned back to `powerOfTwo(1)` which can now build its result and return it back to `powerOfTwo(2)`. One by one all the function calls that were waiting for the partial results of the recursive calls can now build a value and return it back to their callers, all the way up to the start, which ultimately gets the final result.

## 2 Factorial

Let's see another example, very similar to the previous one. The *factorial* of a positive integer number $n$ is the product of all the integer numbers from 1 to $n$. For example:

$$\begin{array}{lcll} 5! & = & 5*4*3*2*1 & = 120 \\ 4! & = & 4*3*2*1 & = 24 \end{array}$$

From the previous example, notice that we can also write the following:

$$5! = 5 * 4!$$

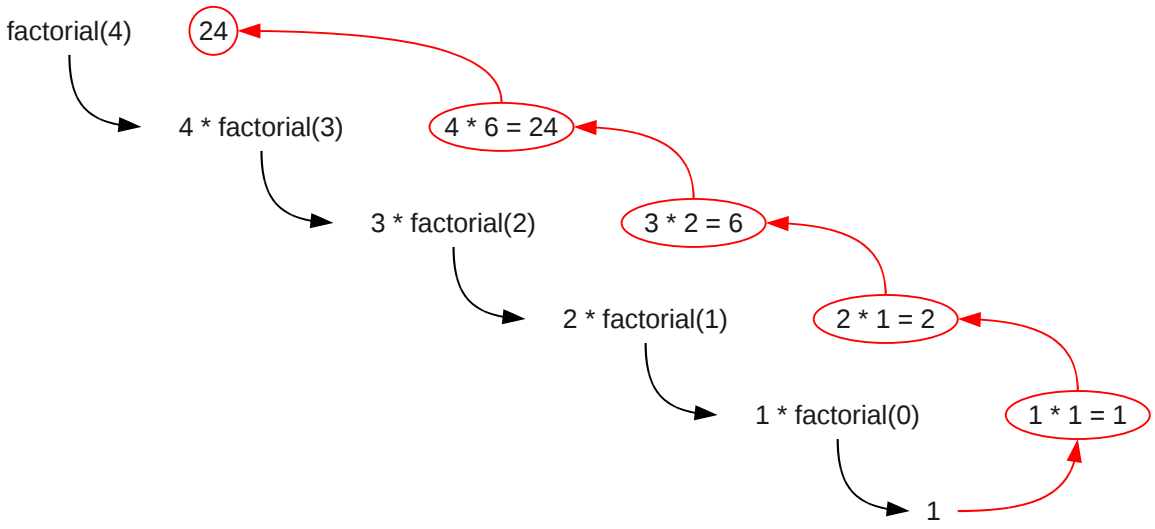
This suggests us that factorial can be defined recursively. By definition, the factorial of 0 is 1.

$$\begin{array}{lcll} 0! & = & 1 & \textbf{base case} \\ N! & = & N * (N-1)! & \textbf{recursive case} \end{array}$$

This is the Java code that recursively calculates the factorial of a positive integer number:

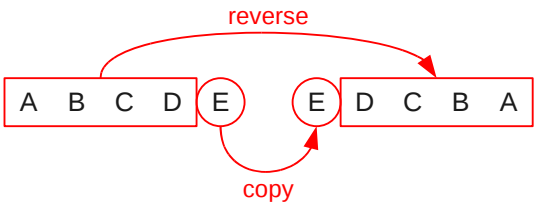
```
// Calculates the factorial of a number using recursion
public static int factorial(int n) {
    // base case: 0! = 1
    if (n == 0) {
        return 1;
    } else {
        // recursive case: n! = n * (n-1)!
        return n * factorial(n -1);
    }
}
```

Once again, we can see how the function works by looking at the execution trace of a call to `factorial(4)`, whose result is 24.




## 3 Reverse of a string

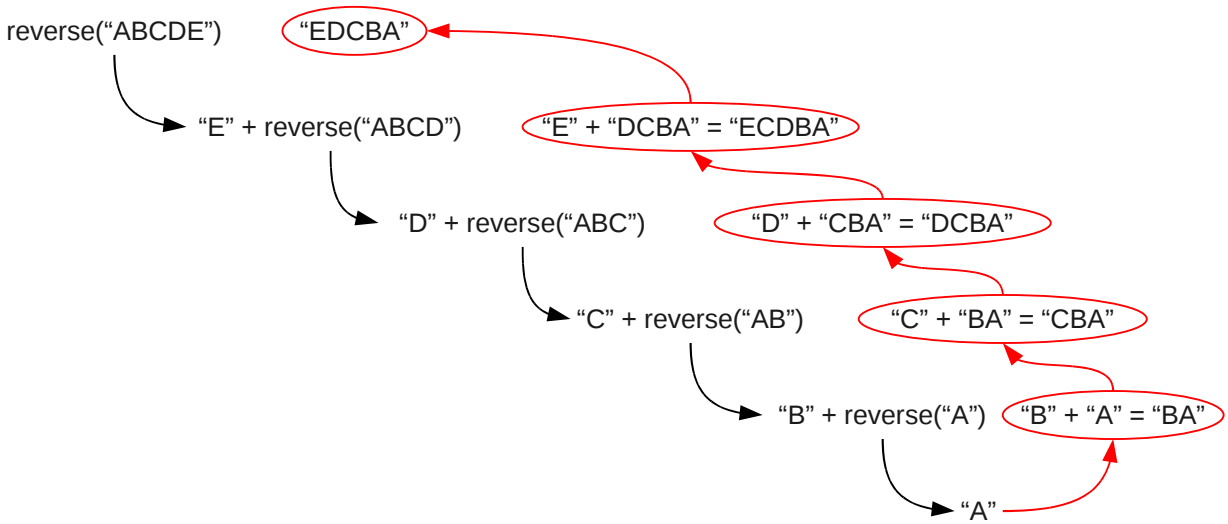
Consider the following problem: given a string, generate a new string with the same characters in reverse order. For example, the reverse of ABCDE is EDCBA. We can think about this problem recursively, by observing that (1) the reverse of an empty string and of a string with only one character is just the string itself (base case); and (2) the reverse of a generic string with more than one character is the concatenation of the last character of the original string and the reverse of the first portion of the original string, i.e., the substring with everything but the last character (recursive case).




The following Java code reverses a string, using the recursive approach just described.

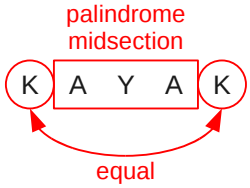
```
// Reverses a string using recursion
public static String reverse(String s) {
    // base case: the reverse of an empty string and the reverse
    // of a string with a single character is the string itself
    if (s.length() <= 1) {
        return s;
    } else {
        // recursive case: the reverse of a string with more than one
        // character is the concatenation of the last character of the
        // original string and the reverse of the first portion of it
        return s.substring(s.length()-1) + reverse(s.substring(0, s.length()-1));
    }
}
```

Let's trace a call to `reverse('ABCD')`.




## 4 Palindromes

Recursion can be a natural way to define and solve the problem of checking whether a string is a *palindrome*. A palindrome is a string that is equal to its reverse. For example, the word "kayak" is a palindrome. Notice that (1) an empty string and a string with less than one character are both palindromes (base case); and (2) a generic string with more than one character is a palindrome only if its midsection (i.e., the substring obtained by removing first and last character) is a palindrome, and its first and last characters are equal.




The following Java code implements the recursive check for palindromic strings.

```
// Checks whether a string is a palindrome using recursion
public static boolean isPalindrome(String s) {
    // base case: the empty string and the string with
    // only one character are both palindromes
    if (s.length() <= 1) {
        return true;
    } else if (s.charAt(0) == s.charAt(s.length()-1)
               && isPalindrome(s.substring(1, s.length()-1))) {
        // recursive case: a string with equal first and last characters,
        // and with a palindrome middle section (everything but its first
        // and last characters), is a palindrome
        return true;
    } else {
        return false;
    }
}
```

Let's trace a call to `isPalindrome('KAYAK')`.