**<Recursive Algorithms(재귀 알고리즘)>**

**Recursion**

Have you ever seen a set of Russian dolls? At first, you see just one figurine, usually painted wood, that looks something like this:



You can remove the top half of the first doll, and what do you see inside? Another, slightly smaller, Russian doll!



You can remove that doll and separate its top and bottom halves. And you see yet another, even smaller, doll:



And once more:



And you can keep going. Eventually you find the teeniest Russian doll. It is just one piece, and so it does not open:



We started with one big Russian doll, and we saw smaller and smaller Russian dolls, until we saw one that was so small that it could not contain another.

What do Russian dolls have to do with algorithms? Just as one Russian doll has within it a smaller Russian doll, which has an even smaller Russian doll within it, all the way down to a tiny Russian doll that is too small to contain another, we'll see how to design an algorithm to solve a problem by solving a smaller instance of the same problem, unless the problem is so small that we can just solve it directly. We call this technique **recursion**.

Recursion has many, many applications. In this module, we'll see how to use recursion to compute the factorial function, to determine whether a word is a palindrome, to compute powers of a number, to draw a type of fractal, and to solve the ancient Towers of Hanoi problem. Later modules will use recursion to solve other problems, including sorting.

**The factorial function**

For our first example of recursion, let's look at how to compute the factorial function. We indicate the factorial of n by n!. It's just the product of the integers 1 through n. For example, 5! equals 1⋅2⋅3⋅4⋅5 or 120. (Note: Wherever we're talking about the factorial function, all exclamation points refer to the factorial function and are not for emphasis.)

You might wonder why we would possibly care about the factorial function. It's very useful for when we're trying to count how many different orders there are for things or how many different ways we can combine things. For example, how many different ways can we arrange n things? We have n choices for the first thing. For each of these n choices, we are left with n-1 choices for the second thing, so that we have n⋅(n−1) choices for the first two things, in order. Now, for each of these first two choices, we have n-2 choices for the third thing, giving us n⋅(n−1)⋅(n−2) choices for the first three things, in order. And so on, until we get down to just two things remaining, and then just one thing remaining. Altogether, we have n⋅(n−1)⋅(n−2)⋯2⋅1 ways that we can order n things. And that product is just n! (n factorial), but with the product written going from n down to 1 rather than from 1 up to n.

When we're computing n! in this way, we call the first case, where we immediately know the answer, the **base case**, and we call the second case, where we have to compute the same function but on a different value, the **recursive case**.

**Properties of recursive algorithms**

Here is the basic idea behind recursive algorithms:

To solve a problem, solve a subproblem that is a smaller instance of the same problem, and then use the solution to that smaller instance to solve the original problem.

When computing n! we solved the problem of computing n! (the original problem) by solving the subproblem of computing the factorial of a smaller number, that is, computing (n-1)! (the smaller instance of the same problem), and then using the solution to the subproblem to compute the value of n!.

In order for a recursive algorithm to work, the smaller subproblems must eventually arrive at the base case. When computing n!, the subproblems get smaller and smaller until we compute 0! You must make sure that eventually, you hit the base case.

For example, what if we tried to compute the factorial of a negative number using our recursive method? To compute (-1)!(−1)!left parenthesis, minus, 1, right parenthesis, !, you would first try to compute (-2)!(−2)!left parenthesis, minus, 2, right parenthesis, !, so that you could multiply the result by -1−1minus, 1. But to compute (-2)!(−2)!left parenthesis, minus, 2, right parenthesis, !, you would first try to compute (-3)!(−3)!left parenthesis, minus, 3, right parenthesis, !, so that you could multiply the result by -2−2minus, 2. And then you would try to compute (-3)!(−3)!left parenthesis, minus, 3, right parenthesis, !, and so on. Sure, the numbers are getting smaller, but they're also getting farther and farther away from the base case of computing 0!0!0, !. You would never get an answer.

Even if you can guarantee that the value of nnn is not negative, you can still get into trouble if you don't make the subproblems progressively smaller. Here's an example. Let's take the formula n! = n \cdot (n-1)!n!=n⋅(n−1)!n, !, equals, n, dot, left parenthesis, n, minus, 1, right parenthesis, ! and divide both sides by nnn, giving n! / n = (n-1)!n!/n=(n−1)!n, !, slash, n, equals, left parenthesis, n, minus, 1, right parenthesis, !. Let's make a new variable, mmm, and set it equal to n+1n+1n, plus, 1. Since our formula applies to any positive number, let's substitute mmm for nnn, giving m! / m = (m-1)!m!/m=(m−1)!m, !, slash, m, equals, left parenthesis, m, minus, 1, right parenthesis, !. Since m = n+1m=n+1m, equals, n, plus, 1, we now have (n+1)! / (n+1) = (n+1-1)!(n+1)!/(n+1)=(n+1−1)!left parenthesis, n, plus, 1, right parenthesis, !, slash, left parenthesis, n, plus, 1, right parenthesis, equals, left parenthesis, n, plus, 1, minus, 1, right parenthesis, !. Switching sides and noting that n+1-1 = nn+1−1=nn, plus, 1, minus, 1, equals, n gives us n! = (n+1)! / (n+1)n!=(n+1)!/(n+1)n, !, equals, left parenthesis, n, plus, 1, right parenthesis, !, slash, left parenthesis, n, plus, 1, right parenthesis. This formula leads us to believe that you can compute n!n!n, ! by first computing (n+1)!(n+1)!left parenthesis, n, plus, 1, right parenthesis, ! and then dividing the result by n+1n+1n, plus, 1. But to compute (n+1)!(n+1)!left parenthesis, n, plus, 1, right parenthesis, !, you would have to compute (n+2)!(n+2)!left parenthesis, n, plus, 2, right parenthesis, !, then (n+3)!(n+3)!left parenthesis, n, plus, 3, right parenthesis, !, and so on. You would never get to the base case of 0. Why not? Because each recursive subproblem asks you to compute the value of a larger number, not a smaller number. If nnn is positive, you would never hit the base case of 0.

We can distill the idea of recursion into two simple rules:

1. Each recursive call should be on a smaller instance of the same problem, that is, a smaller subproblem.

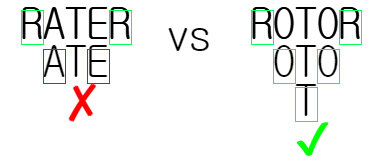
2. The recursive calls must eventually reach a base case, which is solved without further recursion.

Let's go back to the Russian dolls. Although they don't figure into any algorithms, you can see that each doll encloses all the smaller dolls (analogous to the recursive case), until the smallest doll that does not enclose any others (like the base case).

**Using recursion to determine whether a word is a palindrome**

How can you use recursion to determine whether a word is a palindrome? Let's start by understanding what's a base case. Consider the word a. It's a palindrome. In fact, we don't have to think of palindromes as actual words in the English language (or whatever language you'd like to consider). We can think of a palindrome as just any sequence of letters that reads the same forward and backward, such as xyzyzyx. We call a sequence of letters a string. So we can say that any string containing just one letter is by default a palindrome. Now, a string can contain no letters; we call a string of zero letters an empty string. An empty string is also a palindrome, since it "reads" the same forward and backward. So now let's say that any string containing at most one letter is a palindrome. That's our base case: a string with exactly zero letters or one letter is a palindrome.

So here's how we can recursively determine whether a string is a palindrome. If the first and last letters differ, then declare that the string is not a palindrome. Otherwise, strip off the first and last letters, and determine whether the string that remains—the subproblem—is a palindrome. Declare the answer for the shorter string to be the answer for the original string. Once you get down to a string with no letters or just one letter, declare it to be a palindrome. Here's a visualization of that for two words that we discussed:



How would we describe that in pseudocode?

* If the string is made of no letters or just one letter, then it is a palindrome.
* Otherwise, compare the first and last letters of the string.
* If the first and last letters differ, then the string is not a palindrome.
* Otherwise, the first and last letters are the same. Strip them from the string, and determine whether the string that remains is a palindrome. Take the answer for this smaller string and use it as the answer for the original string.

**Computing powers of a number**

Putting these observations together, we get the following recursive algorithm for computing x^nx

n

x, start superscript, n, end superscript:

The base case is when n = 0n=0n, equals, 0, and x^0 = 1x

0

=1x, start superscript, 0, end superscript, equals, 1.

If nnn is positive and even, recursively compute y = x^{n/2}y=x

n/2

y, equals, x, start superscript, n, slash, 2, end superscript, and then x^n = y \cdot yx

n

=y⋅yx, start superscript, n, end superscript, equals, y, dot, y. Notice that you can get away with making just one recursive call in this case, computing x^{n/2}x

n/2

x, start superscript, n, slash, 2, end superscript just once, and then you multiply the result of this recursive call by itself.

If nnn is positive and odd, recursively compute x^{n-1}x

n−1

x, start superscript, n, minus, 1, end superscript, so that the exponent either is 0 or is positive and even. Then, x^n = x^{n-1} \cdot xx

n

=x

n−1

⋅xx, start superscript, n, end superscript, equals, x, start superscript, n, minus, 1, end superscript, dot, x.

If nnn is negative, recursively compute x^{-n}x

−n

x, start superscript, minus, n, end superscript, so that the exponent becomes positive. Then, x^n = 1 / x^{-n}x

n

=1/x

−n

x, start superscript, n, end superscript, equals, 1, slash, x, start superscript, minus, n, end superscript.

**Improving efficiency of recursive functions**

**Memoization of factorial**

We can use a technique called memoization to save the computer time when making identical function calls. Memoization (a form of caching) remembers the result of a function call with particular inputs in a lookup table (the "memo") and returns that result when the function is called again with the same inputs.

A memoization of the factorial function could look like this:

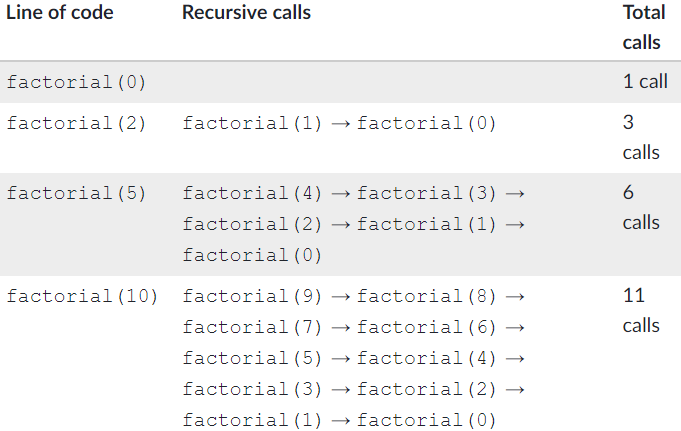
* If n = 0, return 1
* Otherwise if n is in the memo, return the memo's value for n
* Otherwise,
  + Calculate (n - 1)! × n
  + Store result in the memo
  + Return result

This algorithm checks for the input value in the memo before making a potentially expensive recursive call. The memo should be a data structure with efficient lookup times, such as a hash table with O(1)O(1)O, left parenthesis, 1, right parenthesis lookup time.

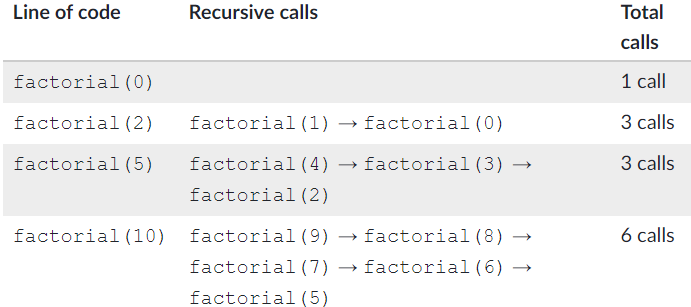
If you'd like to visualize the execution of the memoized algorithm implemented in JavaScript, watch this video(Opens in a new window) or run the visualization yourself(Opens in a new window). Before watching, you may want to challenge yourself to implement the algorithm in the language of your choice.

With memoization implemented, the computer can make fewer total calls over repeated calls to factorial():

before Memoization



after Memoizetion



Memoization makes a trade-off between time and space. As long as the lookup is efficient and the function is called repeatedly, the computer can save time at the cost of using memory to store the memo.

**Memoization of Fibonacci**

The memoized version of the recursive Fibonacci algorithm looks like this:

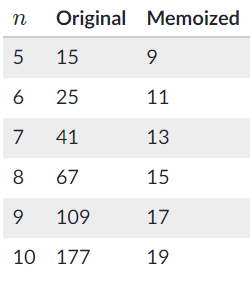
If n is 0 or 1, return n

* Otherwise, if n is in the memo, return the memo's value for n
* Otherwise,
  + Calculate fibonacci(n - 1) + fibonacci(n - 2)
  + Store result in memo
  + Return result

For n=5n=5n, equals, 5, the computer makes 9 calls:

The original version of the algorithm required 15 function calls, so the memoization eliminated 6 function calls.

This table shows the number of calls required for n=5n=5n, equals, 5 up to n=10n=10n, equals, 10:

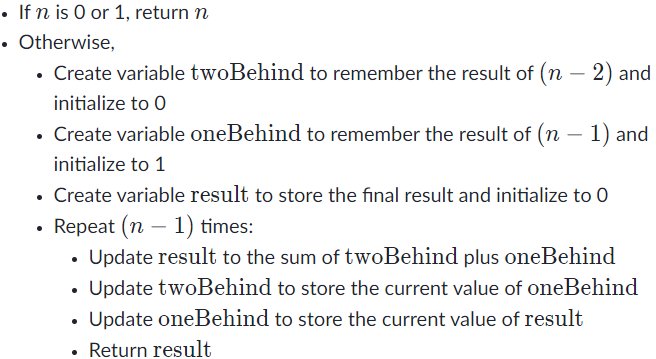


**Bottom-up**

In the case of generating Fibonacci numbers, an iterative technique called the bottom-up approach can save us both time and space. When using a bottom-up approach, the computer solves the sub-problems first and uses the partial results to arrive at the final result.

In the case of generating Fibonacci numbers, an iterative technique called the bottom-up approach can save us both time and space. When using a bottom-up approach, the computer solves the sub-problems first and uses the partial results to arrive at the final result.

A bottom-up approach to Fibonacci number generation looks like this:



This approach never makes a recursive call; it instead uses iteration to sum up the partial results and calculate the number.

The bottom-up algorithm has the same O(n)O(n)O, left parenthesis, n, right parenthesis time complexity as the memoized algorithm but it requires just O(1)O(1)O, left parenthesis, 1, right parenthesis space since it only remembers three numbers at a time.