Visualizing Recursion

Adrian Cisneros · Follow

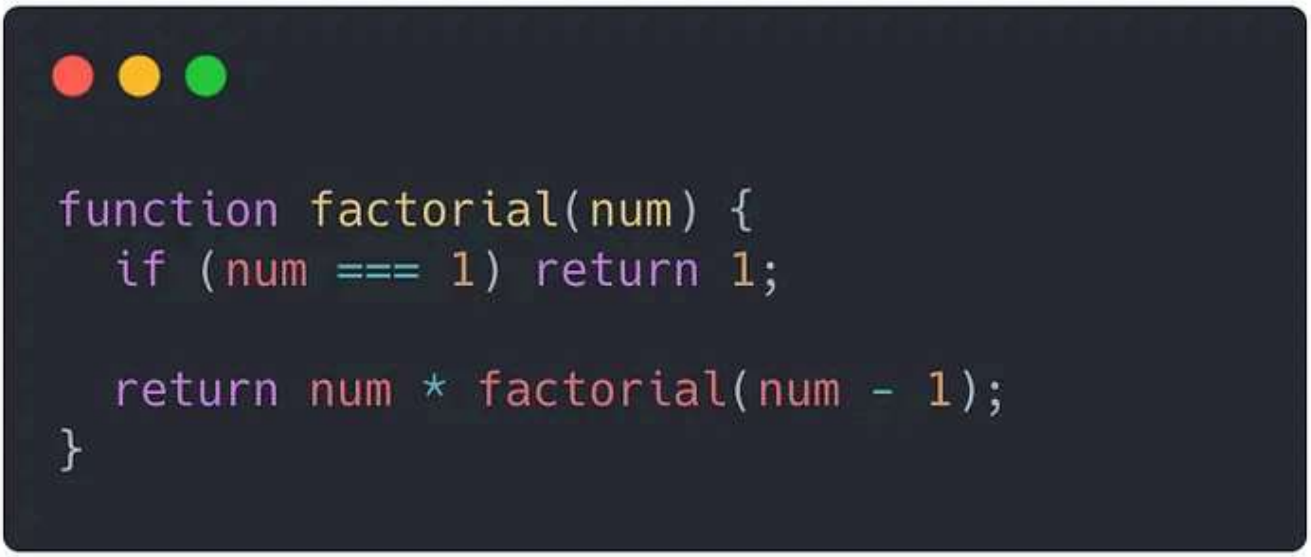
Published in The Startup · 6 min read · Jun 29, 2020

Recursion is a pretty intimidating technique in programming. At least for me it is. However, taking a closer look at what recursion is and what the process looks like has removed some of that intimidation. Hopefully, this can help answer some questions for others and be a nice introduction to recursion.

A recursive function is a function that calls itself. It will continue to call itself until a certain condition, or base case, is met.

The example I will be using to demonstrate this is a factorial. Another name that is fairly intimidating, but is simple. A factorial is the product of a number, n, and every number that is below it. It is written with "!" after the number. So 4! (the factorial of 4) is 24. When we write it out, this is just 4 x 3 x 2 x 1.

Here is the code:



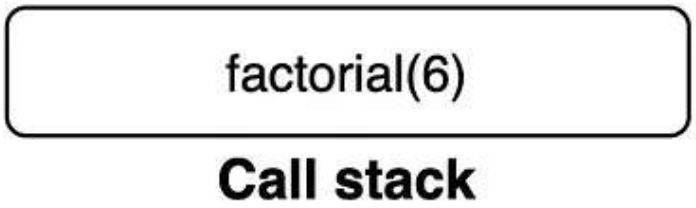
```
function factorial(num) {
  if (num === 1) return 1;

  return num * factorial(num - 1);
}
```

Nothing too crazy here. We call our factorial function and check if we hit out base case, which is that num is equal to 1. If our base case is met, we return 1. Otherwise, we will return the product of our num and our recursively called factorial function. Each time that we call our factorial function, a new function call, or item, will be added , or “pushed”, to the top of the call stack. Each item in the call stack is waiting for the item above it to be resolved in order to resolve itself. The first function call that will be resolved is the one that hits the base case. At that point, the function call that hit the base case will be removed, or “popped”, from the top of the call stack. Now, we have a new top of the call stack and will resolve it. This will continue until we’ve made our way to the bottom of the call stack and we have our final return.

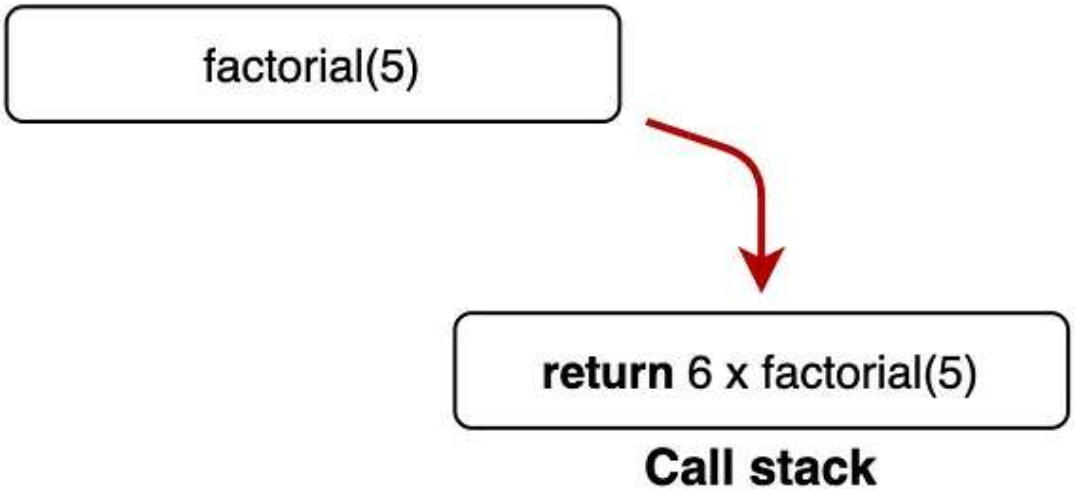
It is difficult to explain in plain text, so this is where our visualization comes in clutch. We will be solving factorial(6) or 6!.

Once we call our function factorial(6), it is pushed to the call stack.



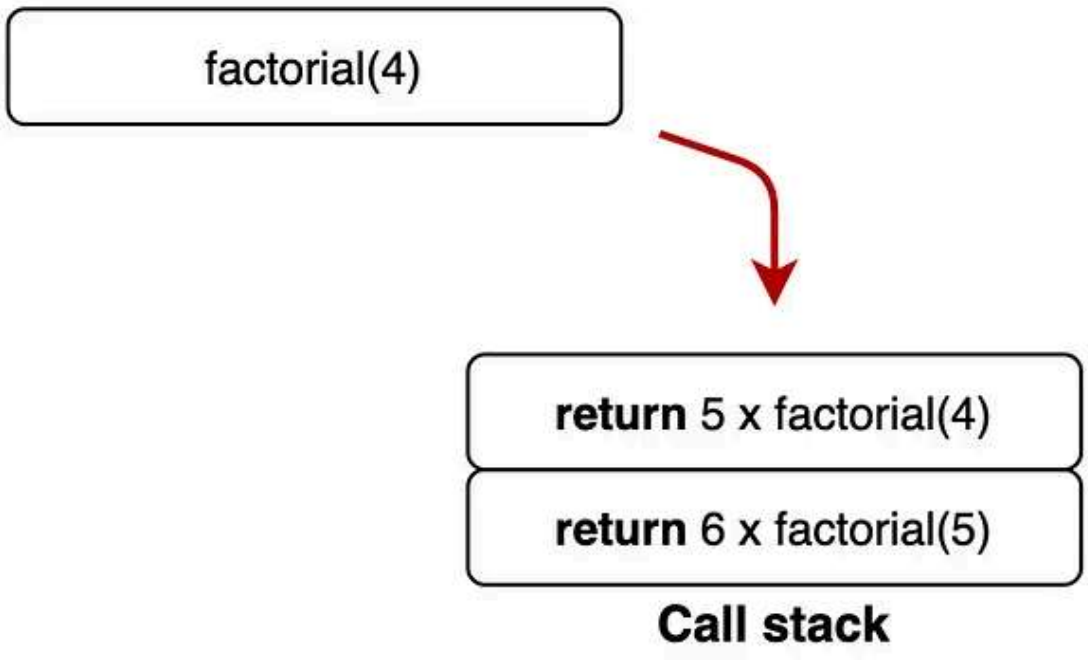
We check if our base case is met, however 6 does not equal 1. So we then need to return the product of 6 and factorial(5). However, because we need factorial(5) to be resolved, the function call for factorial(6) is now just waiting on the call stack.

Once we call factorial(5), we add it to the top of the call stack.



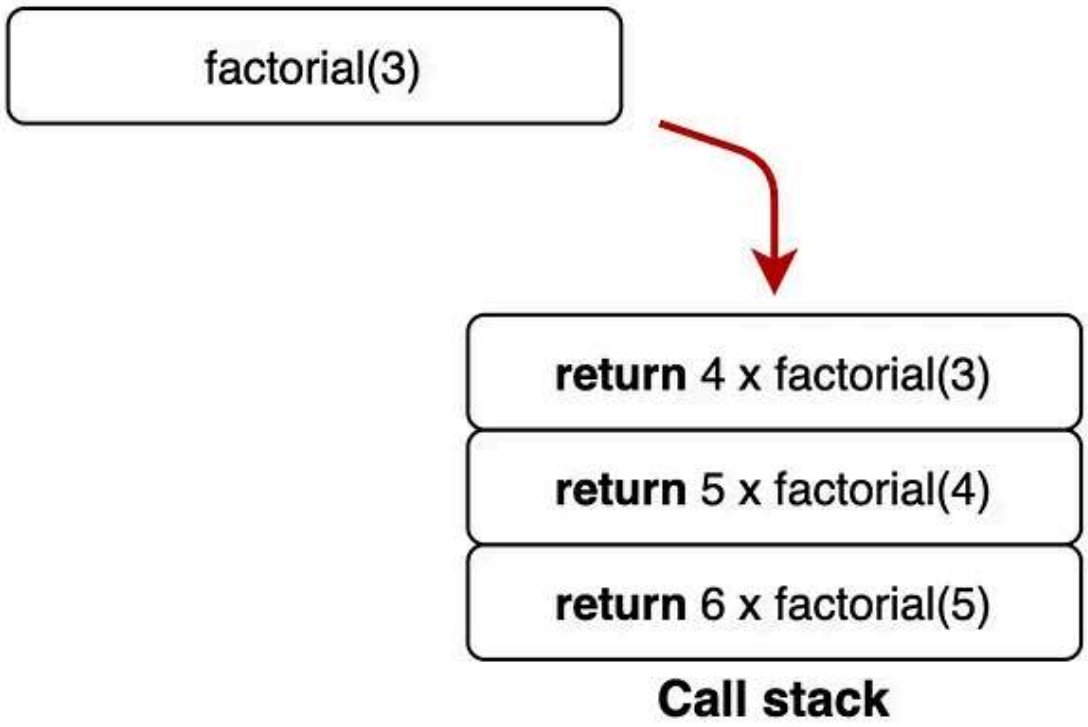
Again, we will check to see if our base case is met. 5 does not equal 1, so it has not been met. We need to return the product of 5 and factorial(4). We need factorial(4) to be resolved before we can return anything, so this is now waiting at the top of our call stack.

Now factorial(4) will be pushed to the top of the call stack.

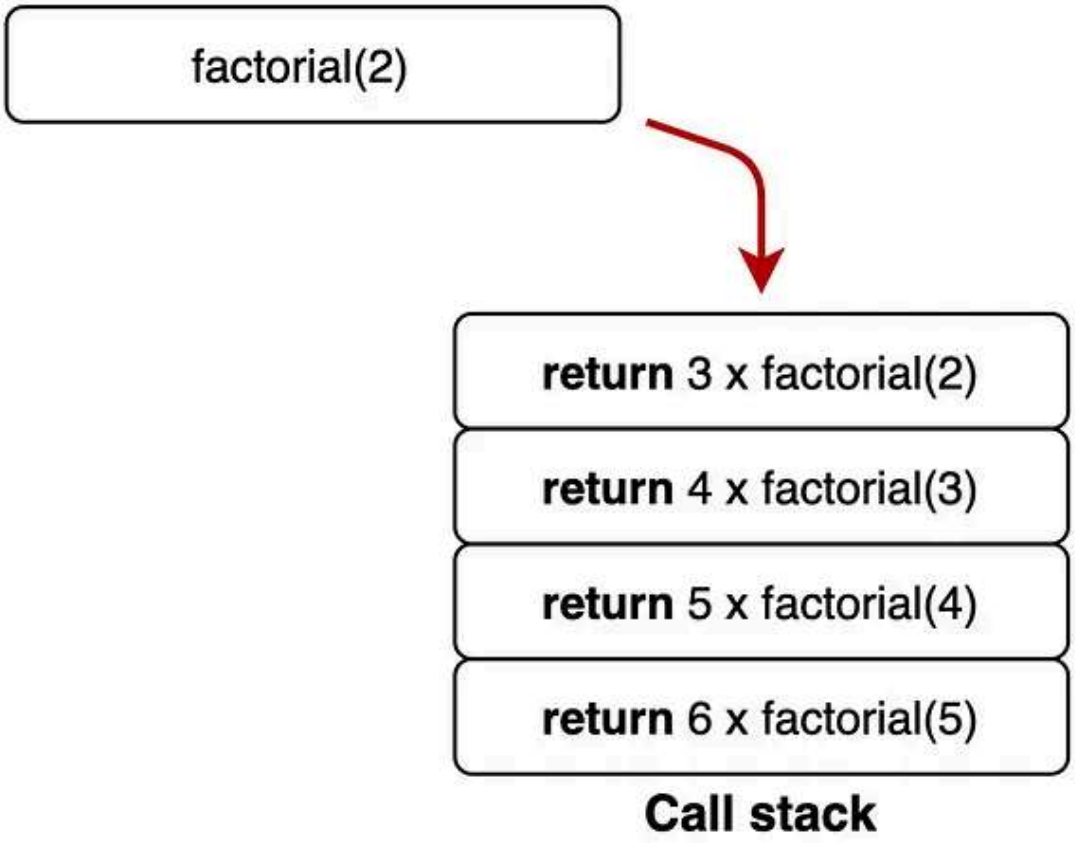


Our num, 4, is not equal to 1, so our base case has still not been met. We will recursively call the factorial function again and return the product of that call and 4.

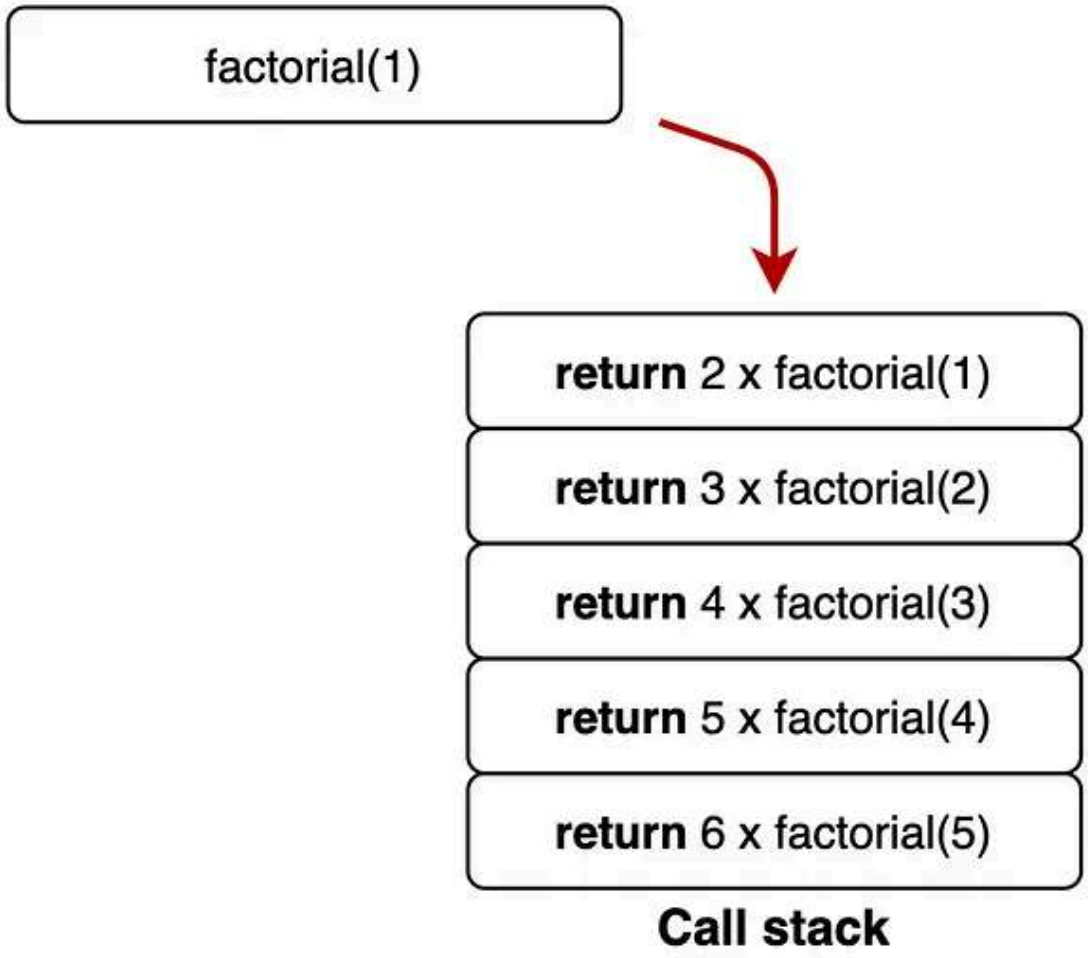
Once factorial(3) is called, it is pushed to the top of the stack.



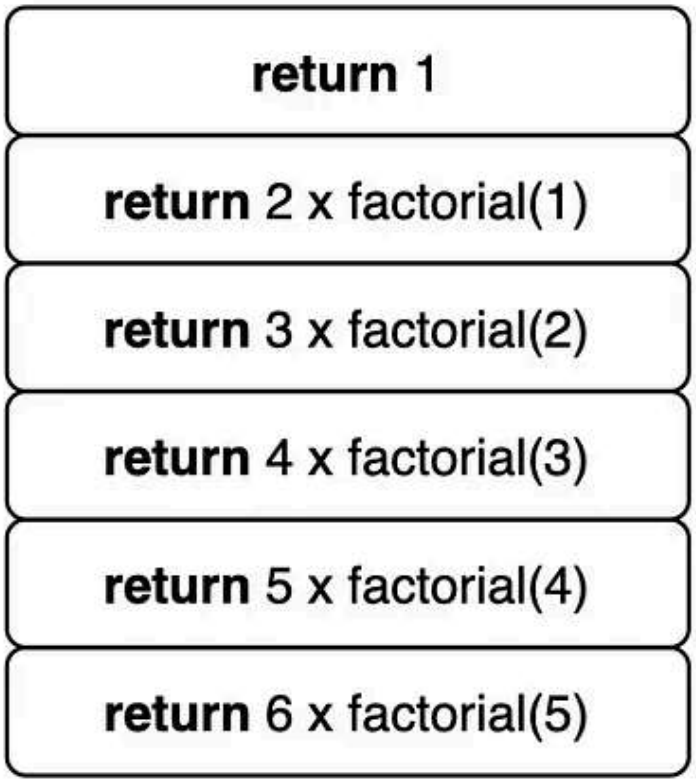
You should start seeing the pattern here. We will continue this process of checking the base case, and either returning 1 or recursively calling our factorial function and pushing it to the top of our stack.



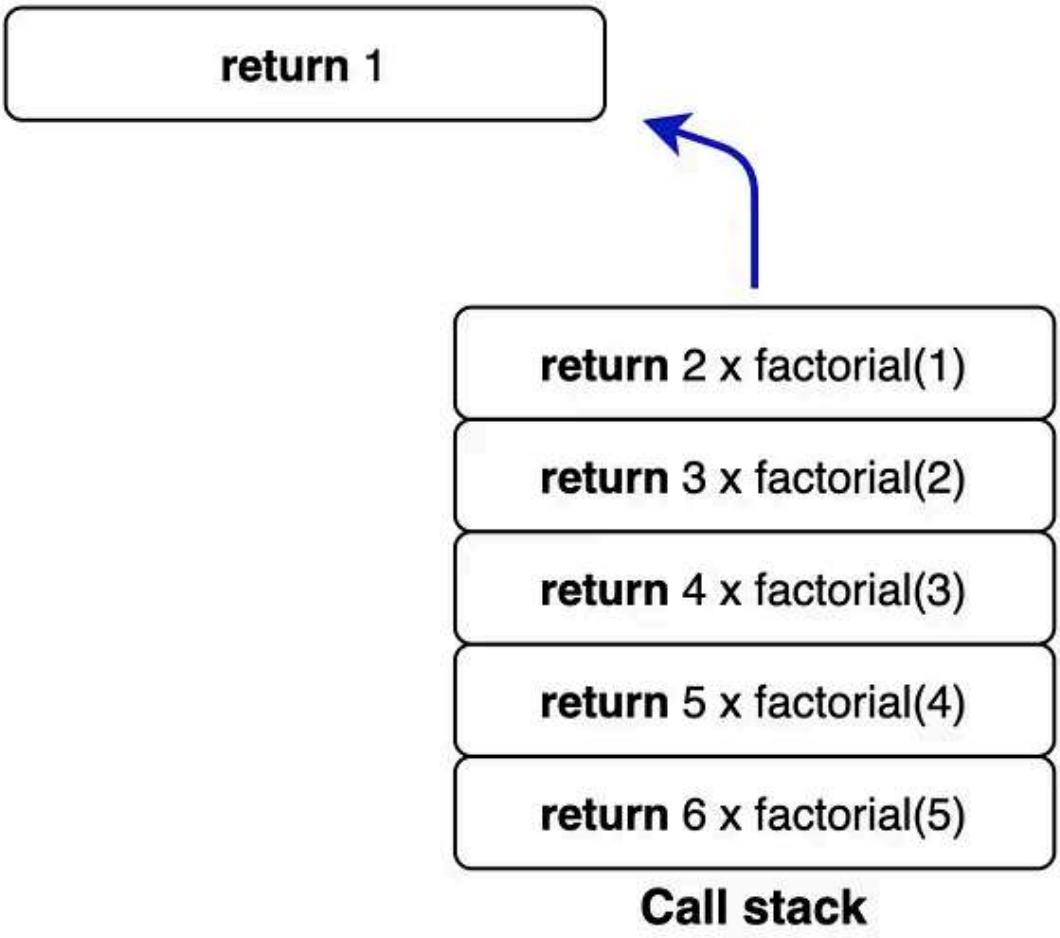
We will need to do this one more time to hit our base case.



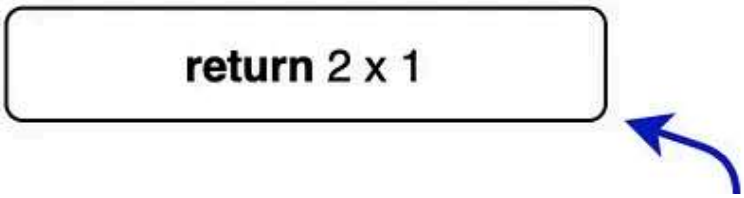
Finally that we have finally that we have met our base case with factorial(1), we can see what our completed call stack now looks like.

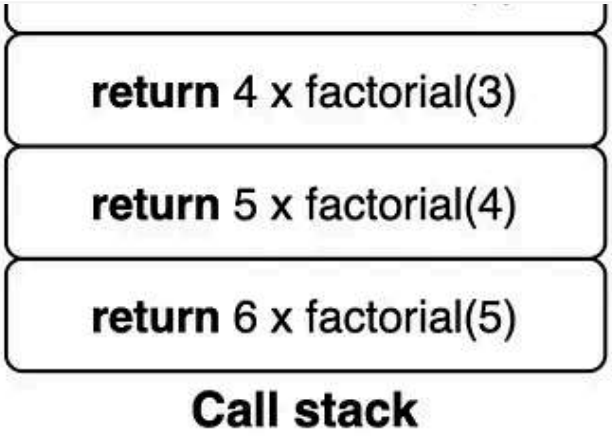


Each item in our call stack is waiting for the item above it to resolve in order to resolve itself. We are basically now going in reverse order. We'll start the process of popping off items from our call stack. Now that we can definitively return a value for factorial(1), in this case that value is 1, we can pop it off the stack.

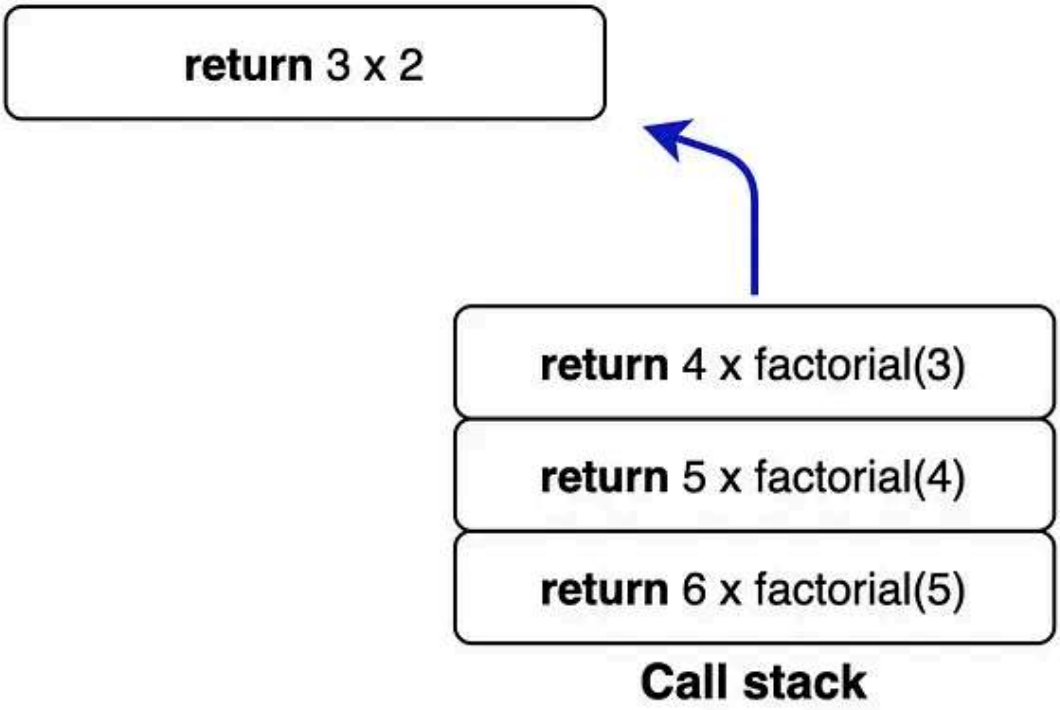


Factorial(1) returned 1. This return value can now be used to resolve the next item in the call stack, factorial(2). And remember in our code, we are returning num * factorial(num -1), which is 2 * 1. We can now pop this off the call stack.

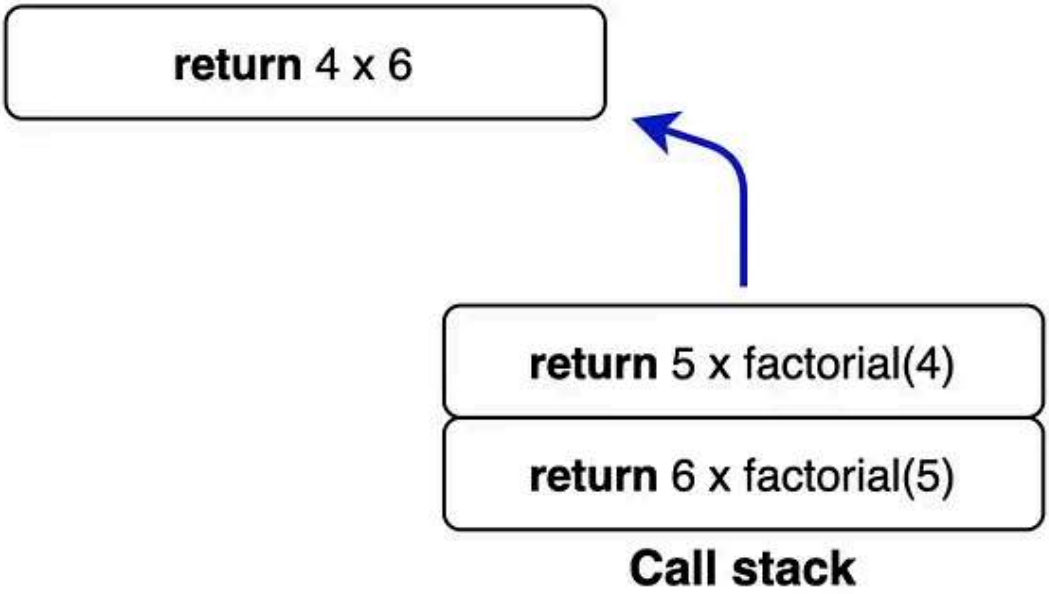




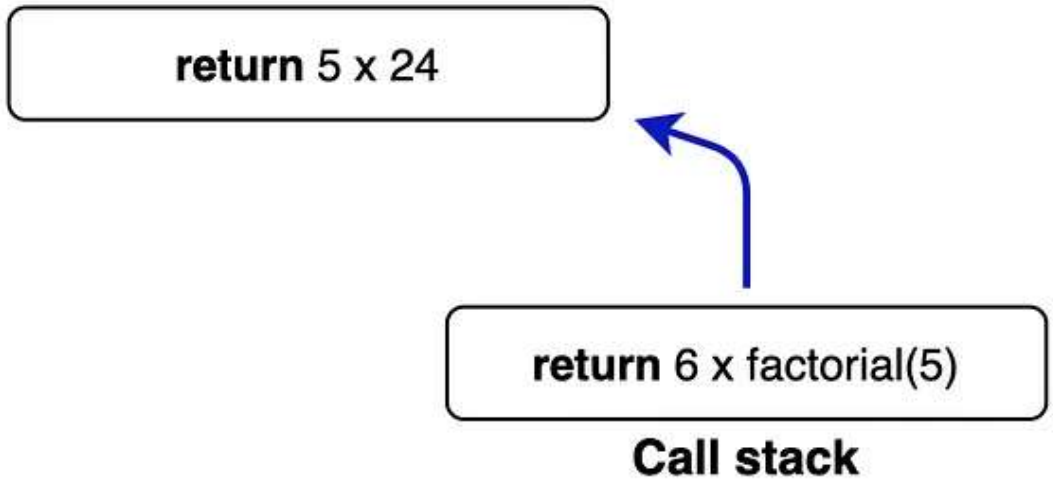
We can use this returned value to now solve the next item in our call stack, which is factorial(3). Num is 3 and factorial(2) returned us 2. So from factorial(3), we are returning 3 * 2, which is 6. We can now pop this off of the call stack.



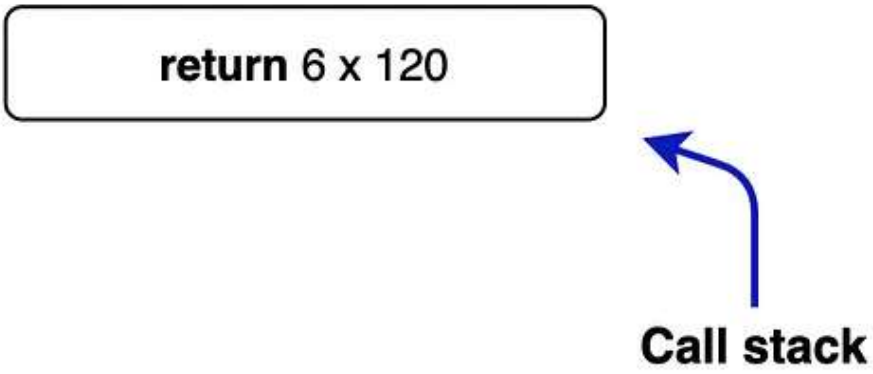
This pattern will continue as we make our way back to the bottom of the stack. We now need a return value from factorial(4), which is 24. This is the product of num, 4, and factorial(3), which returned 6. Factorial(4) can now be popped off from the call stack.



We're almost back at the bottom of the stack. But you should have the hang of it by now. We are now returning the value from factorial(5). We multiply num, 5, and the return value of factorial(4), which is 24, and get 120. Factorial(5) can now be popped off from our stack.



We have now made it to the bottom of our call stack after resolving every other function call that was above it in the stack. Finally, we can return our original function call, which is factorial(6). This will return the product of num, 6, and the value returned from factorial(5), 120. This final function call can be removed from the call stack.



Our call stack is empty and our original function is now resolved and has returned us 720.

As you can see, there can be a lot going on inside of a recursive function. Hopefully this visualization will help you breakdown any recursive functions you create in the future!

If you have any questions or comments, please let me know! You can find me on Twitter at @a_cisneros45.

JavaScript Programming Algorithms Coding

Written by Adrian Cisneros

16 Followers · Writer for The Startup

Software Engineer | React | JS | Ruby

Follow