Understanding recursion in JavaScript is not only considered difficult, recursive function calls in it of itself have a bad rap to its name. Some people go far as to even dub it as the unnecessarily memory intensive and complex version of a for/while loop. Okay, I may have slightly exaggerated the aforementioned statement. Before proceeding, I want to debunk the following beliefs that

1. Recursion is impractical.
2. It is something that I will never fully understand.

First of all, **recursion is not impractical**. It may be difficult to understand at first, but this is why I wrote this post. To bring clarity to this issue. Recursion is a great way of breaking a big problem into smaller, identical problems. Thus, it brings greater clarity to the readers of the code.

Before proceeding however, I recommend that readers brush up and study the following topics.

1. The stack data structure.
2. JavaScript functions.
3. Variable scope and hoisting.

The language of choice for this tutorial will be JavaScript. On demand, I am also more than happy to write out the source code examples in either Java or C++.

Objectives

By the end of this post, I hope that the readers will

Have a deeper understanding of recursion in JavaScript.

Understand cases or problems where a recursive function would be a great tool to use.

Utilize recursion to make newly written/existing code more readable.

Identify and distinguish between good and bad use of recursion.

What is recursion in programming?

In essence, recursion is when a function or a subroutine calls itself repeatedly. All recursive function calls **must have a base case**. A base case is a specific condition that causes the function to return a value instead of calling itself again. Base cases must exist in order to prevent the recursive function from calling itself infinitely. An error will be thrown if the base case is omitted or written incorrectly. I am certain that all programmers who have dabbled with recursive functions have come across this error at some point.

```
Uncaught RangeError: Maximum call stack size exceeded
```

In the previous paragraph, I mentioned the term "incorrect base case". Here, I am referring to a base case that does not cover all the possible user inputs, which may lead to an specific input passing the base case. This will result in an endless recursive function call, resulting in a the call stack overflowing. In the upcoming cases, I am going to cover base cases in more details, using an example.

Base Case Simplified

While thinking about recursion, I suddenly had a AHA moment for explaining what a **base case** is. I am sharing this, because I am hoping that it will help readers understand what a base case really is. Simply, the base case is

> *The answer to a problem at the lowest possible level*

I came up with this myself, so I am kind of proud. Let me explain what this means.

For example we have a factorial function. What is the lowest possible level of a factorial? It is one factorial (1!), which equals 1.

1 factorial (1!) = 1 Note: Factorial(1) is the lowest possible level.

Factorial(2) is Not the lowest possible level

On hindsight, this might only make sense to me, but if it does make sense to you, hooray!

I hope that with this explanation. the concept of base case is somewhat clearer. It is super important, because base cases are literally the building blocks of recursion. You cannot have a recursive function (without resulting in a stack overflow) without a base case.

Function calls are stored on the Call Stack

The reason why I asked viewers to familiarize themselves on the stack data structure is because function calls are stored on the call stack. The call stack is a specific implementation of the stack data structure. It is a LIFO (Last in, first out) data structure, meaning that function calls placed on the top of the call stack are also the first to be popped off. As mentioned in my other blog post, a stack is like a deck of cards. It is more convenient to place and draw cards from the top of the stack of cards.

Some might understand how a stack words from the illustration. However, I also know that visualizing the functions stored will help readers understand with greater clarity. Therefore, in one of the upcoming sections, I will write out a recursive function with a base case. When learning/working on recursive function, examining the call stack by stepping into each function call via a debugging tool is a must. Having good debugging skills here is definitely a plus. If you aren't used to debugging, now is a great time to start getting acquainted to it. Trust me, it will pay dividends in the near future.

What is the Call Stack?

Before heading into analyzing the recursive implementation of the factorial function, we need to have a basic understanding of the **call stack**. As mentioned in the previous section, functions are stored on the call stack. The most important thing to understand when writing recursive function calls is the following piece of information.

> ***In JavaScript function calls are popped off the stack when that function returns a value***

I thought the previous sentence was so important, that it was necessary to emphasize it. Note that even our very own machine manages a call stack or an execution stack. This is actually more advanced and complex and is beyond the scope of this tutorial. For the purposes of this tutorial, keep the quote fresh in your mind.

Tail Call Optimizations and Words of Warning

Note that in JavaScript and any other programming language that does not support proper tail call optimization run the risk of overflowing the call stack when dealing with large data sets. Although ES 2015 now supports tail call optimization in strict mode (according to its specs), as you can see in the browser compatibility table, we have a long way to go until it is supported on all the browsers.

We will get the maximum call stack size exceeded error once again if the data set is too large and the tail call optimization is not made by the compiler.

Not all languages support recursion, depth of stack matters

Recursive function calls example – Factorial Function

Writing code is arguably one of the best ways to understanding recursion. Lets start off with a simple factorial function. By definition, a factorial function (according to wikipedia) is

> *the product of all positive integers less than or equal to n.*

In another words, the factorial of 5 (5!) would be 5 *4 * 3 * 2 * 1 = 120. Firstly, the number 5 is multiplied by itself minus one. This process repeats itself until the number becomes 1. The factorial function involves the same task being broken down into multiple steps. Although the factorial method can be written with a loop, it can also be defined recursively. Why? Because the task itself can be broken down into the same smaller task.

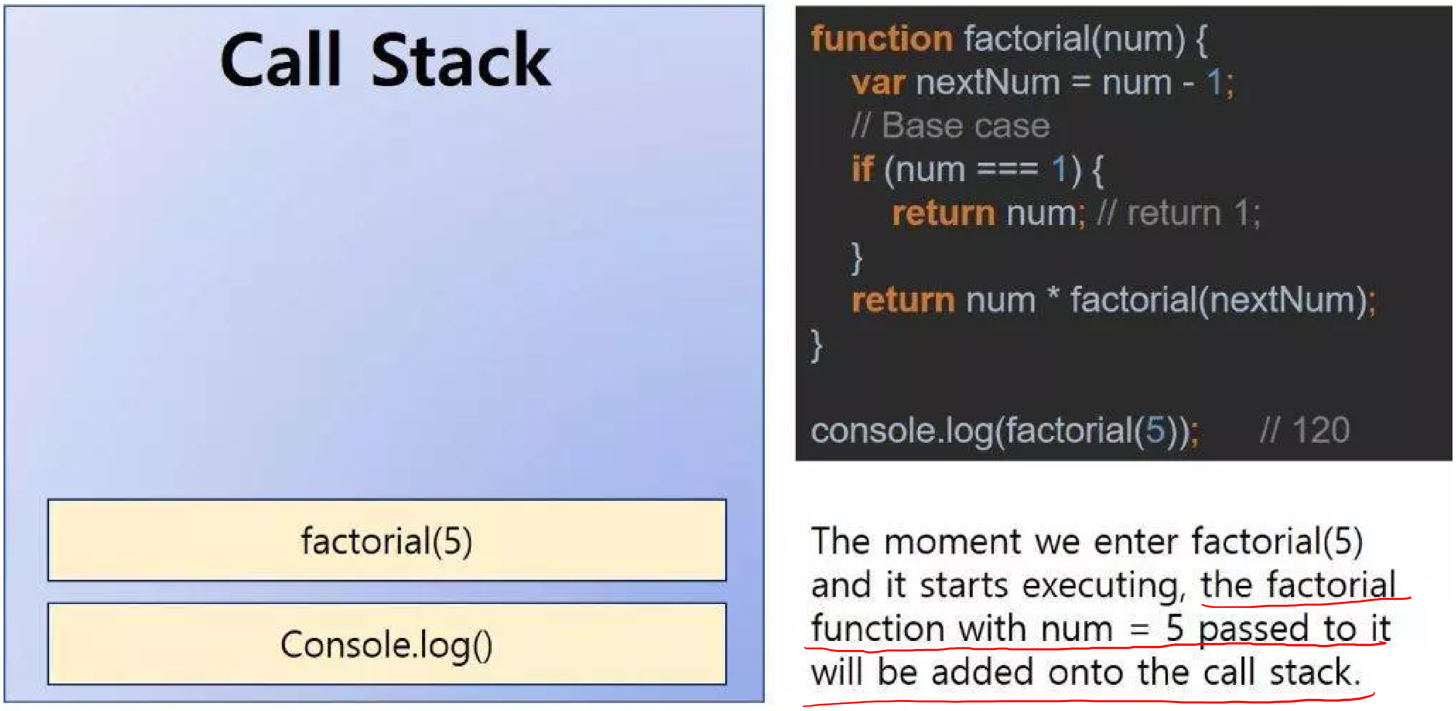
Can you identify what the base case in the example is? **Hint**: I underlined the base case in one of the previous sentences.

```
function factorial(num) {
var nextNum = num - 1;   really clean code
// Base case
if (num === 1) {
return num; // return 1;
}
return num * factorial(nextNum);
}
console.log(factorial(5));      // 120
```

Dissecting and Analyzing the factorial function

Lets examine the code from `console.log(factorial(5));`. First of all, `console.log()`will be pushed onto stack. Afterwards, `factorial(5)` will be executed, and its results will be passed into the `console.log()` function. When we enter `factorial(5)`, the call stack will look something like this.

deleting the call stack



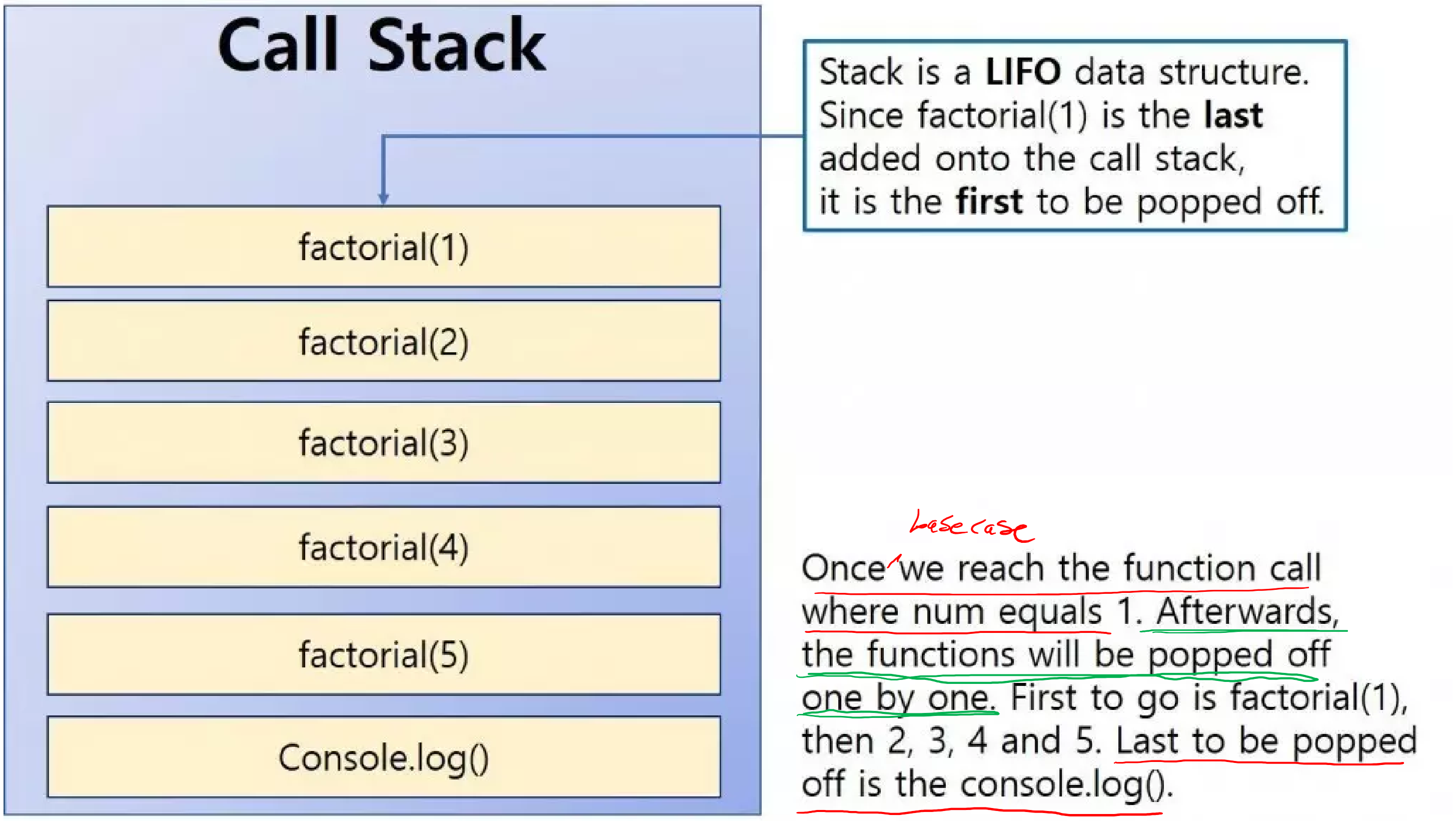
```
function factorial(num) {
   var nextNum = num - 1;
   // Base case
   if (num === 1) {
      return num; // return 1;
   }
   return num * factorial(nextNum);
}

console.log(factorial(5));     // 120
```

The moment we enter factorial(5) and it starts executing, the factorial function with num = 5 passed to it will be added onto the call stack.

Lets skip to line 7 where the return statement is. The factorial function returns `num` (in our example, 5) multiplied by the return value of a recursive function call with 4 being passed in. Essentially, the function returns the following value

```
        return 5 * factorial(4);
```

Because `factorial(4)` is a function, we will push that function call onto the call stack. Now, we will repeat the same process until we reach the base case I.e. when `num`equals 1. By this time, the call stack will look something like this.



Stack is a **LIFO** data structure. Since factorial(1) is the **last** added onto the call stack, it is the **first** to be popped off.

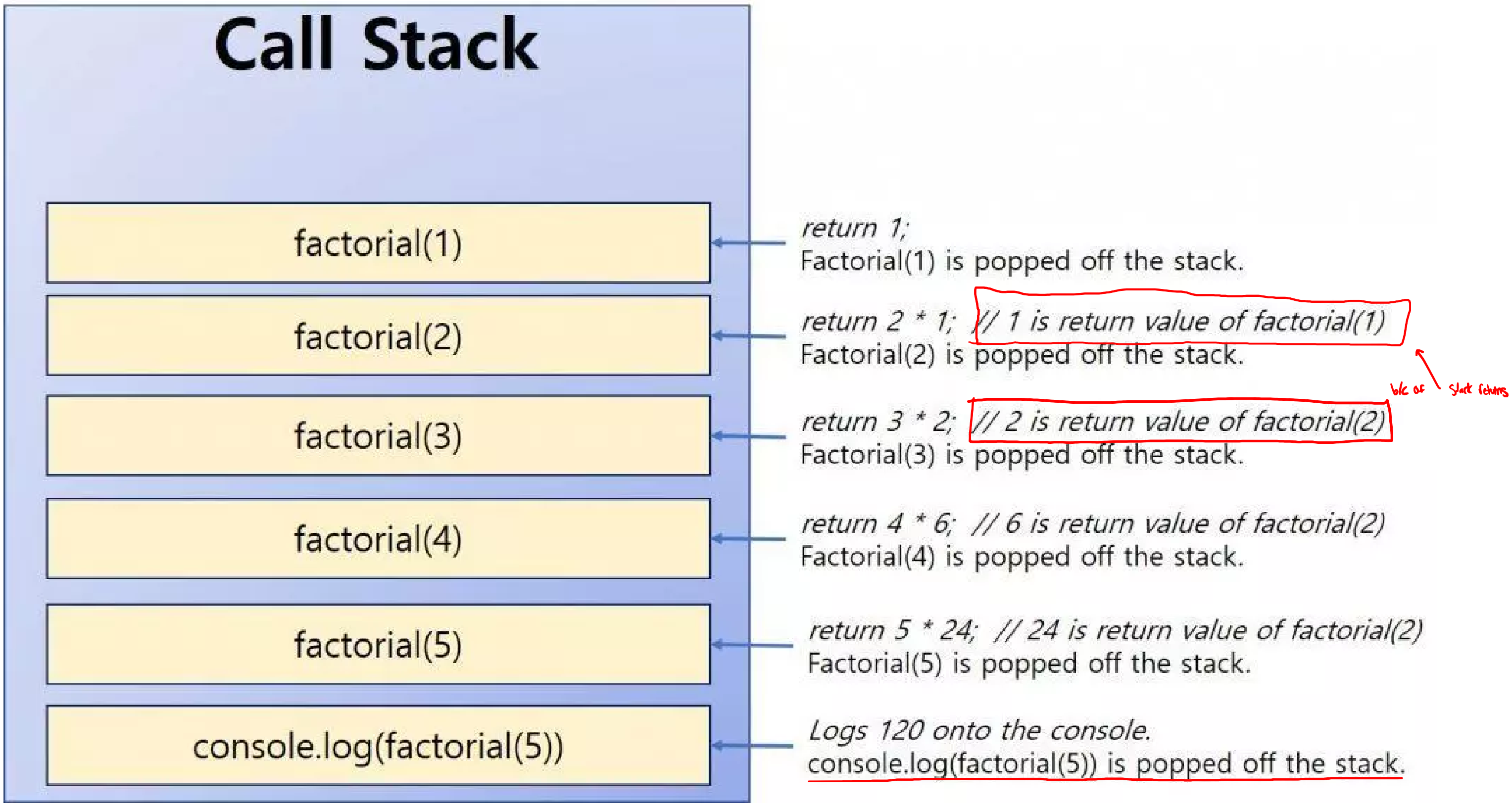
Once base case we reach the function call where num equals 1. Afterwards, the functions will be popped off one by one. First to go is factorial(1), then 2, 3, 4 and 5. Last to be popped off is the console.log().

Once we reach the base case, the function `factorial(1)` returns the value `1`. Therefore, now that we know that `factorial(1)` equals 1, `factorial(2)` also returns a non-function value: `2 * factorial(1)`which is 2 * 1 = 2.

Non-function Value

Moving on, `factorial(3)` returns `3 * factorial(2)`, which equals 6. And so on, until we get to `factorial(5)`, which returns 5 * 24 = 120.

At first, this might sound confusing, so I have done the liberty of adding a visual diagram to help readers digest this information.



return 1;
Factorial(1) is popped off the stack.

*return 2 * 1; // 1 is return value of factorial(1)*
Factorial(2) is popped off the stack.

b/c of stack returns

*return 3 * 2; // 2 is return value of factorial(2)*
Factorial(3) is popped off the stack.

*return 4 * 6; // 6 is return value of factorial(2)*
Factorial(4) is popped off the stack.

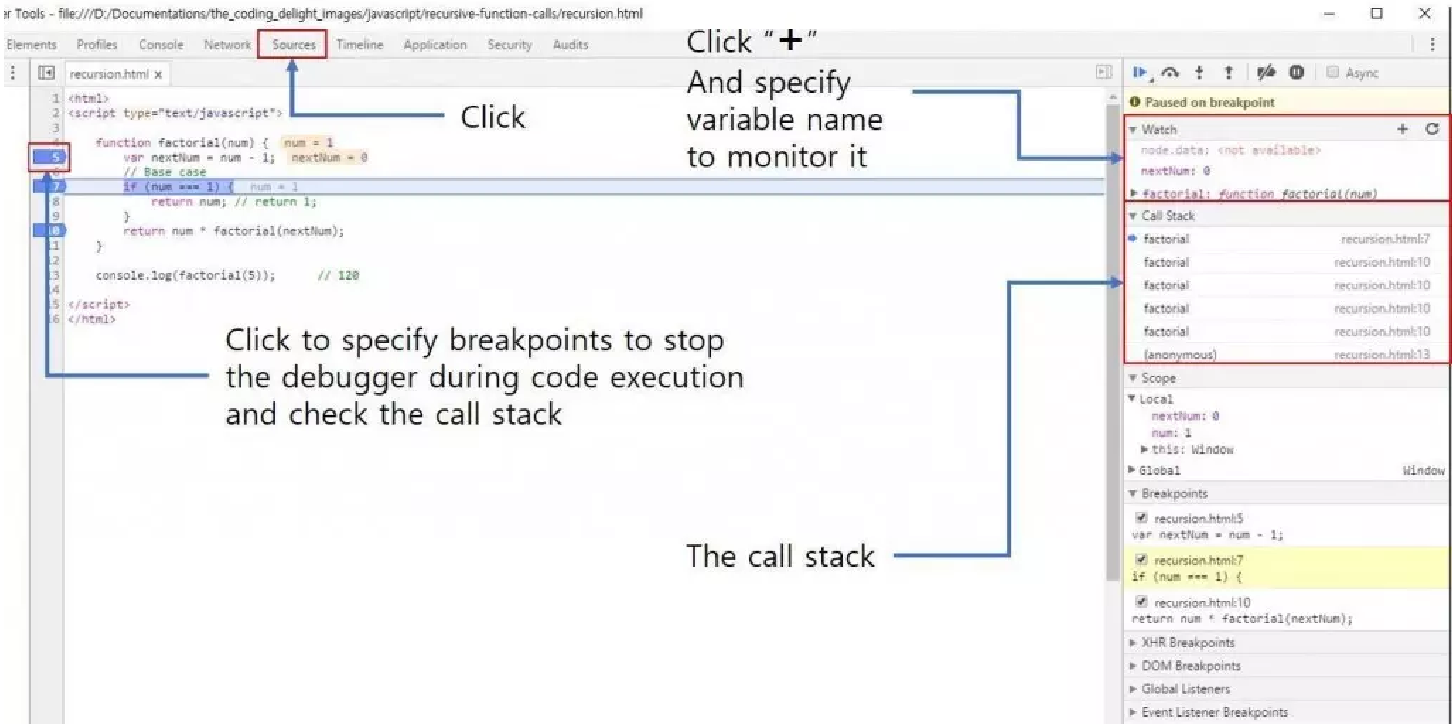
*return 5 * 24; // 24 is return value of factorial(2)*
Factorial(5) is popped off the stack.

Logs 120 onto the console.
console.log(factorial(5)) is popped off the stack.

Call Stack – How can I view it?

For the skeptics or those that are hands-on, you might be asking: Well, how can we actually verify this?

If you are using the chrome web browser (often the simplest way), bring up chrome developer tools (built into chrome, no need to download) by pressing F12 (on Windows). All other major browser vendors should have their own built-in developer tools. On the top tab, you will see menu labels such as Elements, Profiles, Console, Network, Sources, etc. Click on **Sources**. For those using Chrome developer tools, you should see the following window.


r Tools - file:///D:/Documentations/the_coding_delight_images/javascript/recursive-function-calls/recursion.html
Elements
Profiles
Console
Network
Sources
Timeline
Application
Security
Audits
recursion.html ×
Async

```
<html>
<script type="text/javascript">

    function factorial(num) {  num = 1
        var nextNum = num - 1;  nextNum = 0
        // Base case
        if (num === 1) {  num = 1
            return num; // return 1;
        }
        return num * factorial(nextNum);
    }

    console.log(factorial(5));      // 120

</script>
</html>
```

Click

Click "+"
And specify
variable name
to monitor it

Click to specify breakpoints to stop
the debugger during code execution
and check the call stack

The call stack

```
Paused on breakpoint
▼ Watch
  node.data: <not available>
  nextNum: 0
▶ factorial: function factorial(num)
▼ Call Stack
➡ factorial                     recursion.html:7
  factorial                     recursion.html:10
  factorial                     recursion.html:10
  factorial                     recursion.html:10
  factorial                     recursion.html:10
  (anonymous)                   recursion.html:13
▼ Scope
▼ Local
    nextNum: 0
    num: 1
  ▶ this: Window
▶ Global                                   Window
▼ Breakpoints
  ☑ recursion.html:5
  var nextNum = num - 1;
  ☑ recursion.html:7
  if (num === 1) {
  ☑ recursion.html:10
  return num * factorial(nextNum);
▶ XHR Breakpoints
▶ DOM Breakpoints
▶ Global Listeners
▶ Event Listener Breakpoints
```

Thanks to this neat tool, anybody can view the call stack visually. As you can see, when the recursive function call reaches the condition where `num === 1`, it will return 1. Afterwards, each of the factorial function calls will be popped off the stack as the function calls return. Hopefully, the factorial example was a clear explanation of how recursion works.

Read the Chrome DevTools Overview for more information. If you are a front-end developer, chrome developer tools is definitely something you want to quickly get acquainted with.

How do I determine the base case?

When determining the base case for the recursive function calls, it is important to consider the possible user inputs. Sometimes, it can be wise to impose assumptions to improve the performance. By imposing assumptions, you are leaving it up to the consumer of your function to make certain validation checks.

In the factorial example, the base case has a major flaw. Care to guess what the flaw is? The flaw lies in the following line

```
        if (num === 1)
```

What will input be of this function

While the if statement may seem reasonable at first, what happens if I insert a negative number into the factorial function? If you are curious, be my guest and try it. Needless to say that the call stack will overflow, resulting in an error. Therefore, in this case, it would be prudent for the user to update the base case.

Furthermore, what happens if I insert zero? By definition, 0! = 1, but if we enter zero here, a never ending spiral of recursive function calls can be made. Therefore, a more apt solution may be something like the following.

```
if (num === 1 || num === 0)
```

But wait a minute! We still haven't factored out the possibility of the user passing in negative values. Well, we have two common, possible options:

1. Add another if statement that checks whether number is less than zero, and handle that appropriately.

2. Since a negative factorial in Math is `undefined`, document that a negative integer should not be passed into the factorial function. In another words, consumers of the API should make the check. This will ensure that we are not unnecessarily checking for negative values in each recursive function call.

Good documentation helps

I personally prefer the latter approach in this case, but then again, I don't think the factorial function should be expressed recursively.

Examples of poor use of recursion

In the previous factorial example, I personally think that the recursive approach is not the best approach. Using a simple for loop makes the code much more simpler. Furthermore, the validation to check for negative numbers can be made before the loop.

```
function iterativeFactorial(num) {
if (num < 0) {
throw new Error("Negative factorial results in undefined value.");
}
var result = 1;
while (num > 1) {
result *= num;
```

```
        num--;
        }
        return result;
        }
```

The code above is much more robust. One can also argue that the while loop statement is simple enough to not require any explanation.

So ultimately, it all comes down to asking the following question.

> *Is the code more readable when expressed in recursive format?*

However, we also need to consider the robustness and overall performance of the code. Which leads us into the next section, where we will explore the potential trade-offs between the two traits and how to balance the two seemingly elusive traits.

Readability versus Performance

Readers might be wondering: do I need to consider performance when making a decision? Donald Knuth, a famous computer scientist once said

> *We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.*

My personal answer is **both yes and no**. Before making that change for optimization questions, I advise readers to ask the following questions

1. Does the optimization make my code more readable?
2. By optimizing the code, is the performance gain critical?

If the answer to both question is a solid no, then you know the answer: **NO**! However, if the performance gains are critical, but the optimization makes your code slightly less readable, it may be worth documenting. This is to help the next developer who comes across the code to understand it quickly.

The golden rule here is **Prioritize readability**. Only optimize for performance if it is critical. Sometimes, when working with bad code, making optimizations can improve both performance and readability. If this is the case, don't hesitate and go ham with the optimizations.

Can be defined in it of itself

When is recursion a good choice?

Recursion, by nature, is a repetitive task. Therefore, recursion is optimal in cases where a task is repeated multiple times AND can be defined in it of itself. Consider the binary search tree implementation. Each sub-tree in the binary tree is in it of itself, a binary search tree. This means that the traversal algorithm for the binary search tree is an appropriate candidate for a recursive approach.

Another example would be the merge sort algorithm. In this algorithm, we are breaking up a collection into halves, until they all contain only one element. Afterwards, it is repeatedly compared and merged back into a sorted collection. As you can see, merge sort is conceptually based upon breaking up a big task into smaller, equal processes. Therefore, the merge sort is also an ideal candidate for recursion.

Can merge sort be coded without recursion? Yes, but I believe that because it employs the divide and conquer approach, the recursive approach arguably makes more sense.

> *Recursion is a good choice when the problem at hand, or the data structure can be defined in it of itself.*

Remember what I said in the previous section (regarding readability and performance)? When implementing recursive data structures, we generally prioritize readability, unless we really need the micro optimizations. In another words, unless the micro optimizations make the difference between the algorithm executing flawlessly and crashing on user request, we will prefer the more readable approach.

If we have to add an extra 50 lines to a piece of already fairly complex logic to make it iterative, it generally isn't the best idea.

Please note that a recursive solution is very situation, so please, do not use it aimlessly, especially if the problem can be handled with a simple for loop.

Recursion Practice Exercises

Practice makes perfect. Just understanding how the call stack works and how the recursion example above works is not enough. In order to master recursion, developers need to understand not only how it works, but also the thought process behind recursion. The best way to practice is by implementing recursive function calls when solving problems.

As you keep practicing, one of the number-one priority should be readability. As a developer, the number one priority should be writing code that is readable, simple and easy to make sense of. This is where the money is at. Because most of the time, developers spend time reading pre-existing code and maintaining it, instead of writing new code.

During my downtime, I will code the solution to each of these exercises and upload the answers onto GitHub. All the best!

Warm-up Exercises

1. Reverse a string. The recursive function call should return the reversed result of the passed in string. E.g.

```
reverseStr("cowbell") --> "llebwoc"
```

2. Fibonacci number. Get the nth Fibonacci number as the return value. E.g.

```
fibonacci(5) --> 5
fibonacci(10) --> 55
```

> fibonacci(15) --> 610

3. Count the number of reoccurring instances of a digit in a number (E.g. 79092342 has two 9s). For bonus points, create a generator function using closures to create a recursive function using the value passed.
   E.g. Function can generate instances such as

   > count7, count8

   which counts for the digits 7 and 8 respectively.

4. Using recursion, go through a string and remove characters that occur more than once. E.g. passing in `"Troll"` should return `"trol"`. Passing in `"abracadabra"`should return `"abrcd"`.

Intermediate Exercises

Okay, you are starting to get the hang of recursion. The following exercises should leave you scratching your head slightly. No pain, no gain right?

1. Find the greatest common divisor of two numbers.

2. Find the lowest common multiple of two numbers. Assume that the two numbers are greater than or equal to 2.

Advanced Recursion Exercises

The purpose of these exercises is to train your mind to be able to use recursion to solve real world problems. When solving these problems, it often helps to hash out your thoughts vocally, or by drawing activity diagrams.

1. Write a binary search algorithmthat accepts an array. Readers can assume that the passed in array is sorted.

2. Write your own implementation of the merge sort algorithm. As mentioned in a previous section, the merge sort algorithm divides the a big task into smaller tasks. It is one of the better scaling algorithms with a worst case performance of `n log (n)`. The purpose of this exercise is to train the reader to be able to break down a big task

into smaller tasks. Therefore, I highly recommend readers to attempt this problem without looking at the answerbeforehand.

3. Try implementing a recursive data structure. A good example to start off would be implementing the binary search tree from scratch. **Note:** This is quite a challenging exercise, so take your time. Don't beat yourself if you don't get it on your first try.

Recursion JavaScript Exercise Solutions

All recursion exercises and source code are uploaded on GitHub.

Once again, I would like to take a moment to thank all the readers for taking time to read through this post. **Please share with the community if this article helped you understand recursion**. Also feel free to leave a comment, that would be highly encouraging and would spur me on to create even better content. And secondly, while reading, if there is anything that is verbose or difficult to understand, please let me know.

Letting me know what to improve on will ensure that readers get better quality content in the future. Thirdly, feedback on what you, as a reader, liked on the post will let me know what to continue doing (and also hints on how to take it to the next level).

If you are interested in seeing the solutions for the recursion exercises, read on ahead!

Recursion Exercise Walk-through

Due to demands and request, I will be adding a step-by-step guide on each of the recursion exercises. Please note that this will be a continual work in progress and more exercises along with solutions may be added in the future.

Please continue to provide feedback, as it helps me identify which parts I can improve to make the content more comprehensive.

For the sake of clarity and completeness, I will add solutions for both the **iterative and recursive approach**. Please note that this is only a single way of solving the problem. If you solve the problem using a different method, that is perfectly fine as well.

Before going to GitHub and checking up my solution, I recommend readers to attempt each of the problems. If you go straight to the solution without even attempting to solve the problem, you will not improve as much.

Okay, lets head straight into the first exercise.

Note: This section will be a **continual work in progress**. More solutions will be added as I continue to update this post. Currently swamped with a lot of work, but **I am dedicated to make sure that you, the reader, gets great content ASAP**. Expect this post to continuously be updated!

1. Reversing a String Solution

Recursive Approach

Looks fairly simple right? In every recursive problem, I highly recommend starting off by identifying the **task to be repeated**and the **base case**. If you have read every single word up until now, you are probably sick and tired of hearing this from me right?

First of all, the base case. Naturally, when a **string only contains a single character or less** (length of 1)**, it is already reversed right**?

I say less than or equals to because if we limit our base case to a string of length 1, if somebody passes in an empty string, we will never reach the base case, and we will get a stack overflow error. Hope that makes sense ☺

```
        if (str.length <= 1) return str;
```

This is the base case.

Oh, before we go on, I just want to emphasize that there is no right or wrong order of solving the problem. We all think differently, so find the approach that makes the most sense to you. For me, I like to start off with identifying the base case. If that doesn’t work, feel free to try other approaches.

Okay, now we need to write **the task to be repeated**.

Lets identify what we want the base case to return. For example, lets say we are dealing with the following string.

```
var str = "troll";
```

Question to ask here is:

Which character do you want to return from the base case?

If it were me, I would like to return the last character of the string from the base case. So far our function for reversing the string is as follows.

```
function reverseStr(str) {
if (str.length <= 1) {  // base case
return str;
}
}
```

If we want the last character to be at the front, we will naturally want the recursive function call to be at the front of the statement like this right?

```
return reverseStr(str.substr(1)) + str.charAt(0);
```

In this function, we are returning the following to the user.

```
return reverseStr("roll") + "t";
```

We will continue to make the recursive function call until we reach the base case. The call stack will look something like this.

Step 1: initial function call

```
reverseStr("troll");
```

Step 2: first recursive function call

```
return reverseStr("roll") + "t";
```

Step 3: second recursive function call

```
return reverseStr("oll") + "r";
```

Step 4: third recursive function call

```
return reverseStr("ll") + "o";
```

Step 5: fourth recursive function call

```
return reverseStr("l") + "l";
```

Step 6: Hit Base case. Return “l”.

```
return "l";
```

Afterwards, as you know, the results will bubble up as `reverseStr("l")`is popped off the call stack.

Step 7 – 10: Pop off each function off the call stack and return result.

```
return "l" + "l";
```

```
return "ll" + "o";
```

```
return "llo" + "r"
```

```
return "llor" + "t";
```

Hope that this makes sense. If you need me to elaborate, please leave a comment and I will update the post accordingly!

Iterative Approach

The iterative approach is very simple. By now you should know that prioritize readability and only optimize for performance when it is absolutely required.

The Easy Way

Before going on, want to know a super easy way of reversing a string? Here we go.

```
function reverseStr(str) {
return str.split("").reverse().join("");
}
console.log(reverseStr("troll"));  // "llort"
```

Super simple right? Here we are applying the `split()` method to the string `"troll"` . Split accepts two arguments. The first is called the separator and the second argument is known as the limit. For the purposes of this tutorial, we will only be focusing on the separator.

Here, we want each character in the string to be a single element of the resulting array. Therefore, we pass in an empty string.

```
str.split("");
```

In another words, we are converting the string into a character array, that looks like this

```
['t', 'r', 'o', 'l', 'l'];
```

because split returns an array, we are able to call reverse to reverse the contents of our array. After the operation, our array looks like this.

```
['l', 'l', 'o', 'r', 't'];
```

The array is reversed now right? But we don’t want to return an array. We want to return a string to the caller of this method. Therefore, we need to convert the result to a string. A naive way would be to iterate through the array and build up a string like this.

```
var result = "";
for (var i = 0; i < arr.length; i++) {
result += arr[i];
}
```

A more concise way is to utilize the array’s join method. Just as in the `split()` method, our separator will be an empty string, meaning we don’t want to add any additional characters in between the array elements when we convert it to a string.

Hopefully this gives you ideas on how to reverse the string iteratively without using built-in methods.

The Harder Way

Okay, let us actually use a loop to take care of the reversing algorithm, instead of relying on the Array’s built-in `reverse()` method. We will, however, be using `split` to convert the string into a character array and `join` to combine the array back into string format.

```
function reverseStr(str) {
var charArr = str.split('');
// TODO: Write out the reversing logic
}
```

Before proceeding, I want you to ask yourself the following question.

On each iteration, which **two**characters do we swap in order to reverse a string?

Don’t peak at the answers until you have actually thought about it. Try writing out your thought process on a piece of paper. Organize your assumptions and logic in written form.

Once you are ready,

Click here to see the answer

If you don’t need to see the answers, I am assuming you are confident and know how to reverse a string using the iterative approach.

Let’s dive straight into the code.

```
function reverseStr(str) {
var charArr = str.split('');
for (var frontIndex = 0, backIndex = charArr.length - 1;
frontIndex < charArr.length; frontIndex++) {
// Once backIndex is equal to front index, string is reversed.
if (backIndex === frontIndex) {
break;
}
// Perform swap
var temp = charArr[backIndex];
charArr[backIndex] = charArr[frontIndex];
charArr[frontIndex] = temp;
// Decrement index
backIndex--;
}
return charArr.join('');
}
console.log(reverseStr("troll"));  // "llort"
```

If you understood the explanation in “Click here to see the answer“, this code will seem pretty straight forward. Hopefully all of this made perfect sense. If not, please feel free to leave a message and let me know.

2. Fibonacci number Solution

You most likely already know this, but just to be safe I am going to mention it again.

Before solving any problem, building a system, or doing anything in programming, make sure that you first **understand the problembefore writing even a single line of code**.

Lets start off by answering the following question:

> *What is a Fibonacci number?*

The Fibonacci number is a **specific number** from the Fibonacci sequence. The formula for deriving a Fibonacci is as follows

$$x_n = x_{n-1} + x_{n-2}$$

source: mathisfun.com

As quoted in the previous link, the next number in a Fibonacci sequence is equal to the previous number plus the previous number of the previous number.

Lets say we have the following numbers

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ...
```

Try applying the formula above to the sequence. If you take the time and effort to actually go through the sequence and apply the formula to each of the numbers, you will realize that the equation holds true.

The lowest number we can add is one.

If the previous number and the previous number in a Fibonacci sequence was 0 and 0 respectively, we will not get anywhere. We will constantly be adding the number zero, thus ending up nowhere.

Hint: that was a huge giveaway of what the base case for this recursive implementation.

Expected return values for the fibonacci function are as follows

> fibonacci(5) --> 5
>
> fibonacci(6) --> 8
>
> fibonacci(7) --> 13
>
> fibonacci(8) --> 21
>
> fibonacci(9) --> 34
>
> fibonacci(10) --> 55
>
> fibonacci(15) --> 610

Recursive Approach

Okay, now we understand what a Fibonacci sequence is, we can now write our own implementation to solve the problem.

Just as with the previous exercise, we start off by identifying the base case, and which operation we will be repeating.

I gave you a huge hint on what the base case is. Think about it and come back once you have the answer.

The first time you think this through, it can be challenging. But that is why we are here right? To train our minds to process recursively. This will ultimately help you become a better problem solver.

Click here to see the answer

Now that we have our base case, we can start writing the task that we need to repeat. What is the mathematical formula for solving the problem? It is

$$x_n = x_{n-1} + x_{n-2}$$

Simply put, the easiest way to solve this problem would be to simply just apply and return that formula literally. Like the following.

```
return fibonacci(num - 1) + fibonacci(num - 2);
```

In terms of performance, the solution above is terrible and I think even terrible is an understatement. Later on, when I have more time, I will update this post to include potential techniques you can employ to drastically improve the performance of the Fibonacci function. For now, I recommend that you look up memoization in JavaScript.

Iterative Approach

Since we know the logic and formula for the Fibonacci sequence, we are going to jump straight into the code here. If any of the explanations in this section does not make sense, don’t hesitate to let me know.

```
function fibonacciIter(num) {
if (num <= 2) {
return 1;               // First two numbers of fib sequence is equal to
} else {
var result = 0;         // variable holding result
var counter = 2;        // Starting point of iteration
var prevNo = 1;         // prev number aka xn-1
var prevPrevNo = 1;     // prev number of previous number aka xn-2
while (counter < num) {
result = prevNo + prevPrevNo;   // xn = xn-1 + xn-2
var temp = prevNo;              // Temp variable to store value
prevNo = result;                // xn-1 = xn
prevPrevNo = temp;              // xn-2 = xn-1. Make sense right?
counter++;
}
}
```

```
        return result;
```

In the code above, the following statement is acting as our base case. Naturally, the first two numbers are equal to 1, so we return one if `num` is less than or equal to 2.

```
        if (num <= 2) {
        return 1;               // First two numbers of fib sequence is equal to
```

Take note that negative answers will also yield one. So if you want to address this issue, be my guess and add some defensive measures to the code above. It should be fairly straight forward.

The upcoming section is where the real action is.

```
        else {
        var result = 0;         // variable holding result
        var counter = 2;        // Starting point of iteration
        var prevNo = 1;         // prev number aka xn-1. Equals one. Fib sequenc
        var prevPrevNo = 1;     // prev number of previous number aka xn-2. Star
        while (counter < num) {
        result = prevNo + prevPrevNo;   // xn = xn-1 + xn-2
        var temp = prevNo;              // Temp variable to store value
        prevNo = result;                // xn-1 = xn
        prevPrevNo = temp;              // xn-2 = xn-1. Make sense right?
        counter++;
        }
        }
```

I would say that the comments and the code combined together explains the logic, but for the sake of completeness, let us walk through this problem step by step.

Take note that `counter = 2`. This is our starting point. Thus, we need to initialize counter to 2, and also initialize `prevNo` and `prevPrevNo` to 1. Why? Because the first two items in the Fibonacci sequence are 1 and 1.

E.g. **1**, **1**, 2, 3, 5, 8, 13 …

The rest of the variables will be set in the `while` loop.

On each iteration, we are applying the following formula.

$$\mathbf{x_n = x_{n-1} + x_{n-2}}$$

`var temp` is used to temporarily store `prevNo` so that we can override the previous previous number (`prevPrevNo`or $\mathbf{x_{n-2}}$) with the previousNumber (`prevNo` or $\mathbf{x_{n-1}}$). Like the following

$$\mathbf{x_{n-1} = x_{n-2}}$$

Lastly, we are incrementing counter (`counter++`) to iterate n amount of times and reach the condition for terminating the `while` loop.

3. Digit Counter Solution

In this exercise, we are counting the number of times a certain digit occurs in a number. For example, if we are looking for the number of 9s in the number 79092342, the function should return 2, as two nines can be found.

So first of all, lets look at the arguments that the function should receive. I think it should be two. The first argument being the number we will be working with. The second argument should be the number we are looking for.

```
function digitCounter(number, digitToCount)
```

Here is a question for you.

> *Should we assume that the reader appropriately inserts a number that is 0-9 as the digit to count? Or should we add validations in our function?*

Critically assessing and weighing these kind of options is what separates a decent programmer from a good programmer. Here are some factors that you should consider.

What is the cost of adding the validation check?

What are the possibilities of incorrect inputs being passed in?

In this case, the validation check only takes **O(1)** constant time, so I suggest adding it to minimize human error. If however, the time complexity of adding the check ramps up to **O(n)** linear time – checking a whole list, you may want to add assumptions to the function and comment your code so that users pass in the appropriate data.

Here, we can add the following validation check

```
if (digitToCount > 9 || digitToCount < 0) {
throw new Error("digitToCount must be between 0 and 9");
}
```

Or you can be smart and convert first the digit to a string. Afterwards, check the length of the string and see if it does not equal one.

```
// Validation checks
if (typeof digit !== "string") {
digit = digit.toString();
}
if (digit.length !== 1) {
throw new Error("Please enter a single digit.");
}
```

This is quite a nice solution in this case, because we will eventually have to convert the number to a `string` in order to compare its individual digits.

Think about it for a moment. If it is a negative number, the string will have a minimum length of 2. If the number is greater than 9, will have a length of 2. Therefore, only numbers from 0 to 9 will have a length of 1 when converted to a string. Pretty neat right?

Of course, there are other ways of approaching this problem. This is just simply one approach out of many. Therefore, feel free to be as creative as you want.

In the next section, we will examine the recursive approach to the problem.

Recursive Approach

I think you are getting more and more confident with recursion now, so I am going to get straight to the point.

```
function getDigitCount(num, digitToCount) {
// Validation checks. Note that you can also throw an error if digit is
if (typeof digit !== "string") {
digitToCount = digitToCount.toString();
}
if (digitToCount.length !== 1) {
throw new Error("Please enter a single digit.");
}
if (typeof num !== "number") {
throw new Error("num must be of type number");
}
// Keep track of digits
var digitOccurCount = 0;
// Convert number to string
num = num.toString();
var recursiveDigitCounter = function(num, digitToCount) {
// Base case
if (num.length === 0) {
return;
}
var currentNumber = num.charAt(0);
if (currentNumber === digitToCount) {
digitOccurCount++;
}
recursiveDigitCounter(num.substring(1, num.length), digitToCount);
};
recursiveDigitCounter(num, digitToCount);
return digitOccurCount;
```

Note that this is not the most optimal approach. And there is plenty of room for refactoring to improve performance and readability. But the logic should be fairly clear and self-explanatory.

After converting the number to a string, we are using a very similar approach to the approach employed in the **reversing a string exercise**earlier on.

Therefore, I am not going to bother repeating myself.

If you really want me to add explanations, feel free to contact me and let me know. I will follow up as soon as I can.

Iterative Approach

Iterative approach is much more simpler and elegant for this kind of operation. Therefore, I recommend using the iterative approach.

If you think about it, I am sure that you found implementing the recursive approach much more challenging right?

Anyhow, lets get straight into the code.

For the sake of brevity, I am going to remove the validation checks, which are included in the recursive approach. We will be solely focusing on the code inside of the `for` loop.

```
function getDigitCountIter(num, digitToCount) {
var digitOccurCount = 0;
num = num.toString().split('');
var len = num.length;
for (var i = 0; i < len; i++) {
var curNum = Number(num[i]);
if (curNum === digitToCount) {
digitOccurCount++;
}
```

```
        }
        return digitOccurCount;
        }
```

The iterative approach is so simple that we barely have to explain it. All we are doing in the loop is converting the string item back into a number and comparing it. If it is the digit we are counting, increment the counter. Simple right?
Let us move onto the next question.

4. Solution to Removing characters that occur more than once

The first question that you may ask is

> *How do I keep track of the characters that we have seen and check whether they occurred more than once?*

If this is your first time dealing with this kind of problem, it may be potentially daunting. But don't worry, we will examine this problem step by step together.

The big problem to keep track of here is to keep a record of all the characters we encountered. In JavaScript, the simplest way of doing this is storing in a JavaScript object. E.g.

```
        var charMap = {};
```

This might not make any sense right now, but trust me when I say that we are going somewhere.

Because manipulating strings in each iteration via concatenation can be somewhat pricey, we are going to store the characters inside of an array and convert it to a string once we have iterated through the entire string.

When we examine a character in the string, we will be doing the following activities.

```
        var charMap = {};
        var result = [];
```

1. Check whether `charMap` has the key value ‘a’;
   If the character ‘a’ exists, do not add it to `result`.

   If it doesn’t, add new property ‘a’ to `charMap` with value set to true to mark it as having occurred.

Here is what the key checking operation described above looks like as code.

```
if (!charMap.hasOwnProperty(char)) {
charMap[char] = true;  // mark as having occurred
result.push(char);     // push into results
}
```

Although there are other ways of approaching this problem, this solution is the one I think is easiest to understand. Now that we have gone through the meat of the problem, lets take a look at the recursive approach to the problem.

Before we move on, I would like to explain the reason why we are storing repeated characters inside of a JavaScript object as opposed to an array.

Reason for storing Repeated Characters in JavaScript Object

Recursive Approach

Here is the code snippet that we will be working with.

```
function removeReoccurringCharacters(str) {
if (typeof str !== "string") {
throw new Error("Please enter a string!");
}
var charMap = {};
var result = [];
var removeRepeatingChar = function(str) {
var len = str.length;
// base case
if (len === 0) {
return str;
}
```

```
        var firstChar = str.charAt(0);
        if (!charMap.hasOwnProperty(firstChar)) {
        charMap[firstChar] = true;  // mark as having occurred
        result.push(firstChar);     // push into results
        }
        removeRepeatingChar(str.substring(1, len));
        };
        removeRepeatingChar(str);
        return result.join('');
        }
```

Everything here is fairly standard. One thing I would like to point out here, is that because we are working with external, mutable values, which in this case are `charMap` and `result` respectively, our recursive function does not need to return a value.

Note that because `charMap` and `result` are contained within its own scope of `function removeReoccurringCharacters`, we don't have to worry about side effects.

As I mentioned, we are using `array`as a string builder so that we don't have to pay the price of concatenating strings, which can become costly when working with a large string.

Once we have the final string, we are simply going to call `join()` on result to form a string and return that.

Iterative Approach

We pretty much explained the logic, so I am going to paste the code below. One point I would like to mention is that in all the four exercises we've walked through together, all of these solutions are better off implemented iteratively rather than recursively. In terms of BOTH performance and readability.

Remember to critically assess the trade-offs before making a decision.

```
        function removeReoccurringCharactersIter(str) {
        if (typeof str !== "string") {
        throw new Error("Please enter a string!");
```

```
}
var charMap = {};
var result = [];
var charArray = str.split('');
for (var i = 0; i <charArray.length; i++) {
var currentChar = charArray[i];
// Character doesn't exist. Add to result
if (!charMap.hasOwnProperty(currentChar)) {
charMap[currentChar] = true;    // mark as character having occurred
result.push(currentChar);       // push into results
}
// Otherwise, do nothing
}
return result.join('');
}
```

Final Words

And that is it! By now, you should be fairly familiar with JavaScript recursion and understanding how it works behind the scenes. I hope that you learned a lot from reading and working through this tutorial, and that it has given you a boost in your journey towards becoming a better programmer.

Until next time, peace and happy coding!

You probably need to rest your brain if you managed to read this entire post in one seating.