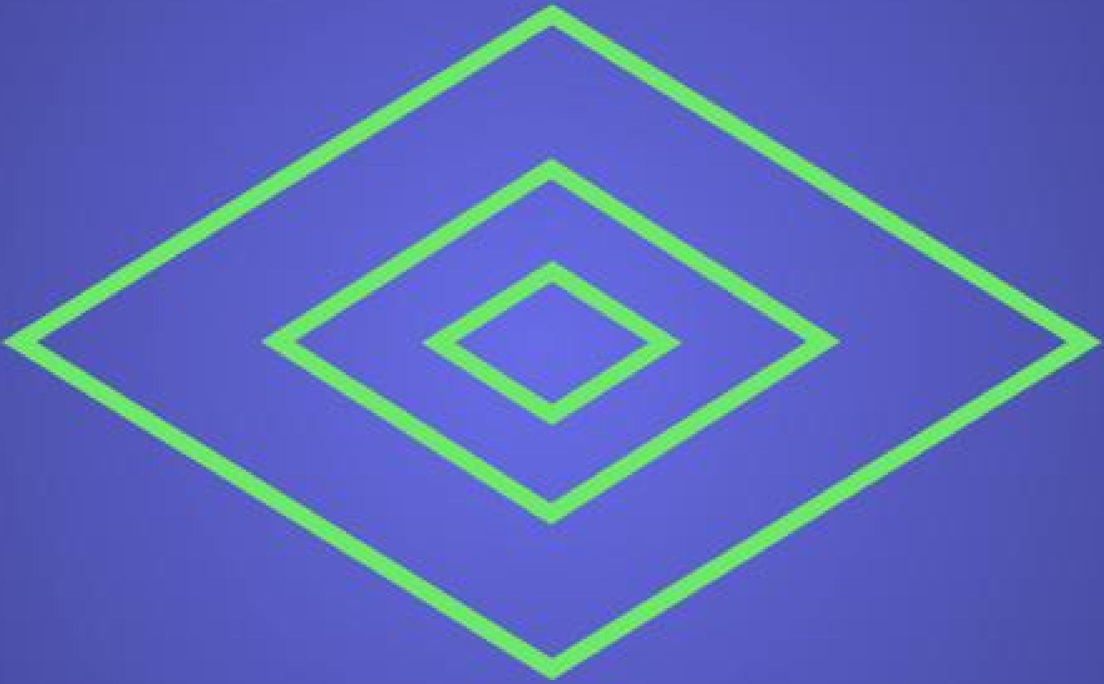
Understanding Recursion For Beginners



Pavle Paunović

Published by:
Pavle Paunović

Table Of Content

Introduction and word of the author

Hello and welcome to the book! In this book, I will teach you how to use recursion and most importantly you will gain an understanding of what recursion is.

Recursion is a hard topic to grasp (I know because it took me a while to understand it).

Recursion is a beautiful concept and it will help you solve many hard problems easily.

I just want you to know that you can do it! You can learn recursion. I will teach you how to to use recursion on arrays, strings, nested objects (trees). I will give you step by step examples, so you can see how recursion works under the hood and effects that have on memory.

In book I will use JavaScript programming language. I will assume that you have basic knowledge of programming.

Just a disclaimer, in the book the function declarations are used. It is easier for beginners to grasp recursion, instead using arrow functions. Arrows functions are used for callbacks.

I hope you enjoy the book!

Pavle Paunović

What is recursion?

A recursive function is a function that it calls itself.
Simple as that!

Let me give you an example:

```
function recursion(n) {
    return recursion(0);
}
```

That right there is recursion! But wait! If you run the code, it will run infinitely number of times (or until you have memory).
(Do not run this code, it will freeze your browser or browser tab, it is just for demonstrating purpose)

We need a way to stop the recursion.

We need some condition that when is met it will stop calling the function.

That condition is called a **base case.**

```
function recursion(n) {
    if (n === 10) { [1]
        return n;
    }
    return recursion(n + 1); [2]
}
```

[1] - This is a **base case**, the place where a function is stopped and returns an **n.**
[2] - Here the **n** is incremented by one.

Look at this, you can **console.log** the **n**, just like in normal loop.

```
function recursion(n) {
    if (n === 10) {
        return n;
    }
    console.log(n);
    return recursion(n + 1);
}
```

It will log numbers until 10. Simple, right?

From while/for loops to recursion

You saw on the last few pages what recursion is. Let’s convert some **while/for** loops to recursion, so you can start using recursion right away instead of normal loops!

```
let i = 10;

while (i >= 0) {
    console.log(i);
    i--;
}

Now, let’s convert it to recursion,

function iterateToZero(n) {
    if (n === 0) {
        return n;
    }
    console.log(n);
    return iterateToZero(n-1);
}
```

Simple!

Now, let's start adding multiple arguments!

```
for (let i = 0; i <= 100; i++) {
    console.log(i);
}
function iterateToSomeNum(n, k) {
    if (n > k) {
        return k;
    }
    console.log(n);
    return iterateToSomeNum(n + 1, k)
}

iterateToSomeNum(0, 100)
```

Here we have a recursive function with two parameters. You can put as many parameters as you want. Parameter **k** is always the same number. And we increment **n** by one.

Let us see another example:

```
function recurseOddNumbers(n, k) {
    if (n === k) {
        return n;
    }
    if (!(n % 2 === 0)) {
        console.log(`Odd ${n}`)
    }
    return recurseOddNumbers(n + 1, k);
}
```

I hope you understand now recursion on the basic level, and you know how to use it instead of basic loops. Good job! Let us keep moving forward!

Chapter 1: Recursion and arrays

Let us see how we can use recursion with arrays!
First, let us sum the array! Pretty simple, but pay attention!

Array sum

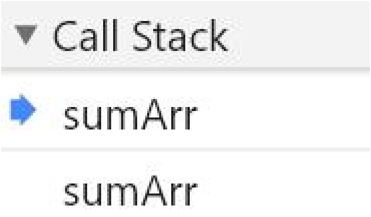
```
function sumArr(arr) {
    if (arr.length === 0) {
        return 0;
    }
    return arr[0] += sumArr(arr.splice(1));
}
sumArr([1,2,3,4]) // 10
```

How come this works? Let me introduce you to new concept called the **Stack**.

Stack

A stack is a **LIFO (Last In First Out)** data structure.
This means that every recursive function call is put on top of the stack and removed from the stack when the recursive call is done!
Let us analyze step by step **sumArr** function

--------- **Step 1** -> 1 + sumArr([2,3,4]);

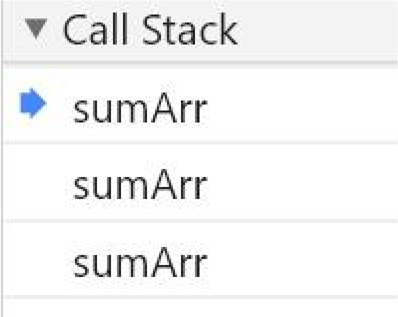


▼ Call Stack

➡ sumArr

sumArr

--------- **Step 2** -> 1 + 2 + sumArr([3,4]);



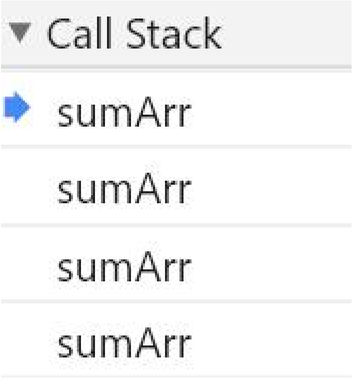
▼ Call Stack

➡ sumArr

sumArr

sumArr

--------- **Step 3** -> 1 + 2 + 3 + sumArr([4]);



▼ Call Stack

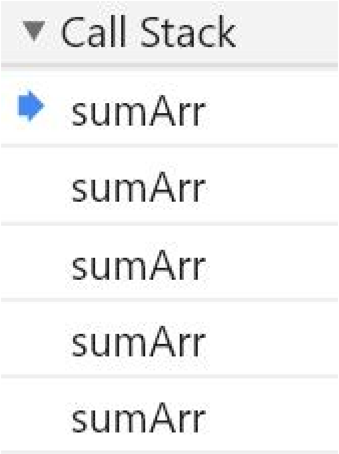
➡ sumArr

sumArr

sumArr

sumArr

--------- **Step 4** -> 1 + 2 + 3 + 4 + sumArr([]);



▼ Call Stack

➡ sumArr

sumArr

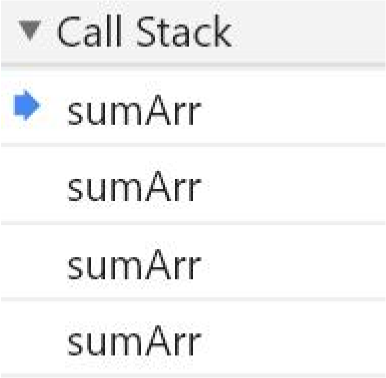
sumArr

sumArr

sumArr

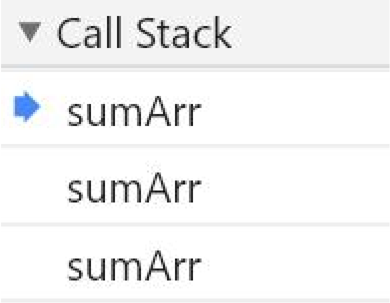
Okay, we got to the end of the stack, what now? The recursion has reached **base case**. Now the stack unwinds! Remember how I told you that **stack** is **LIFO (Last In First Out). Watch!**

1 + sumArr([2,3,4])
1 + 2 + sumArr([3,4]);
1 + 2 + 3 + sumArr([4])
1 + 2 + 3 + 4 + sumArr([])
---------------- 0 + 4



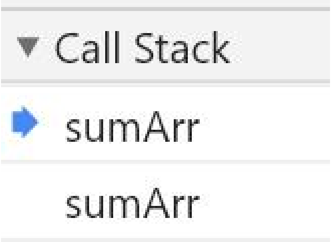
▼ Call Stack
sumArr
sumArr
sumArr
sumArr

----------- 4 + 3



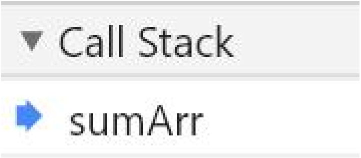
▼ Call Stack
sumArr
sumArr
sumArr

-------7 + 2



▼ Call Stack
sumArr
sumArr

---- 9 + 1



▼ Call Stack

➡ sumArr

---10

Yea! We got the right summed number. This is how recursion works behind the scene. Very nice, right?
I want you to see how we can iterate array with recursion.

We **splice** array items by one, meaning array length will change the more computer puts functions on the stack.

www.pavlepaunovic.com