Unit 3: Algorithms

•••

Recursion

Roadmap

- What is recursion?
- Why use recursion?
- Types of Recursion

What is Recursion?

A function is recursive if...

It calls itself within its own definition!

It calls itself within its own definition!

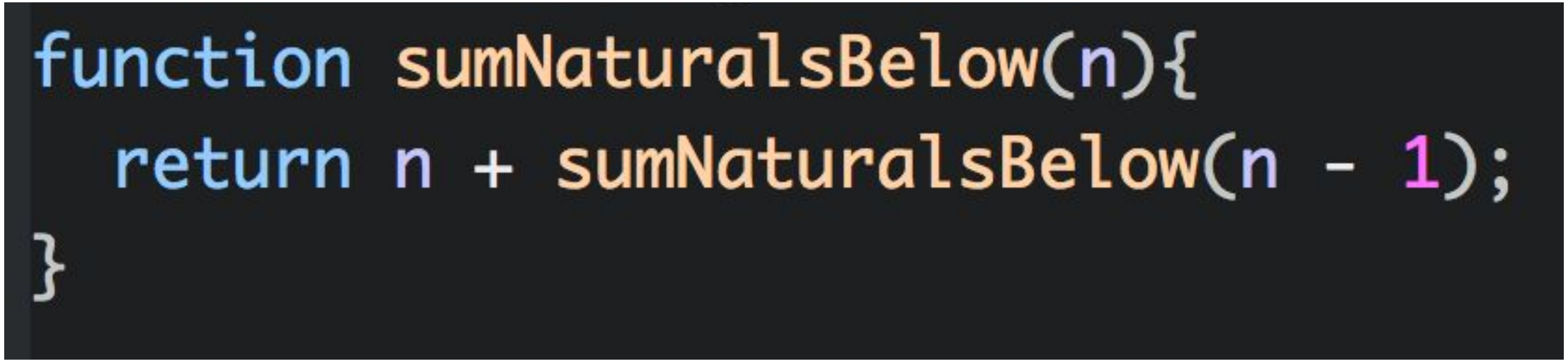
It calls itself within its own definition!

It calls itself within its own definition!

It calls itself within its own definition!

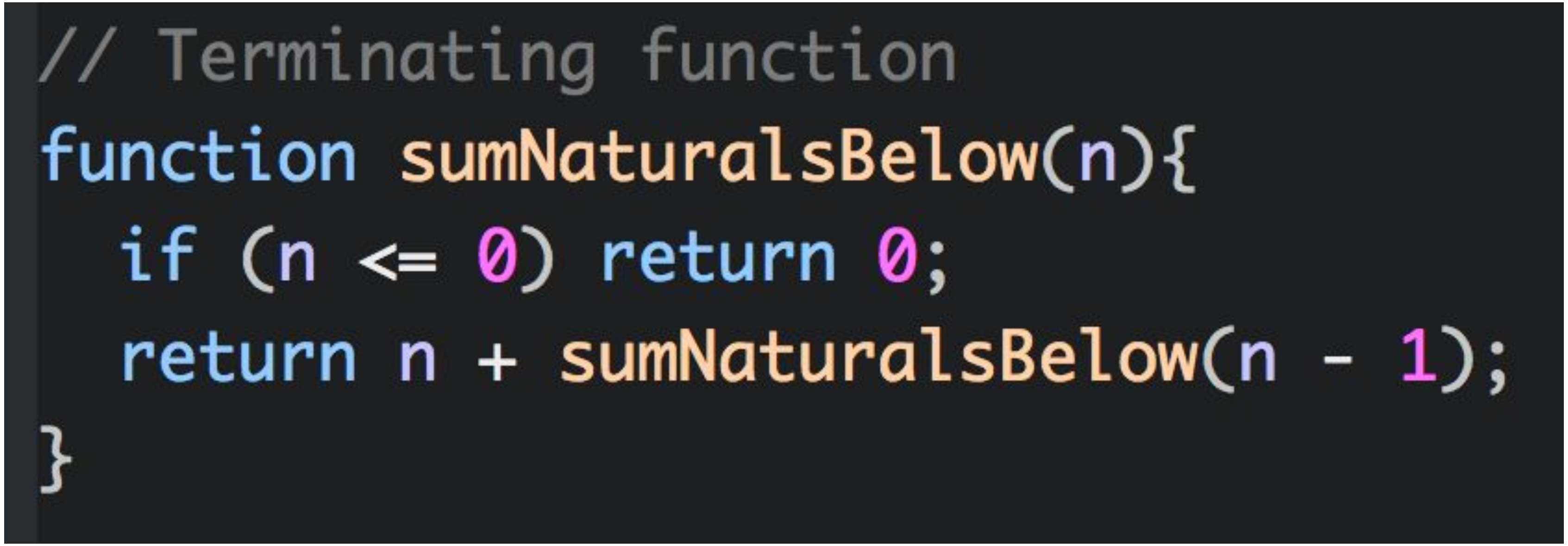
It calls itself within its own definition!

Consider the following function...



```
function sumNaturalsBelow(n){
  return n + sumNaturalsBelow(n - 1);
}
```

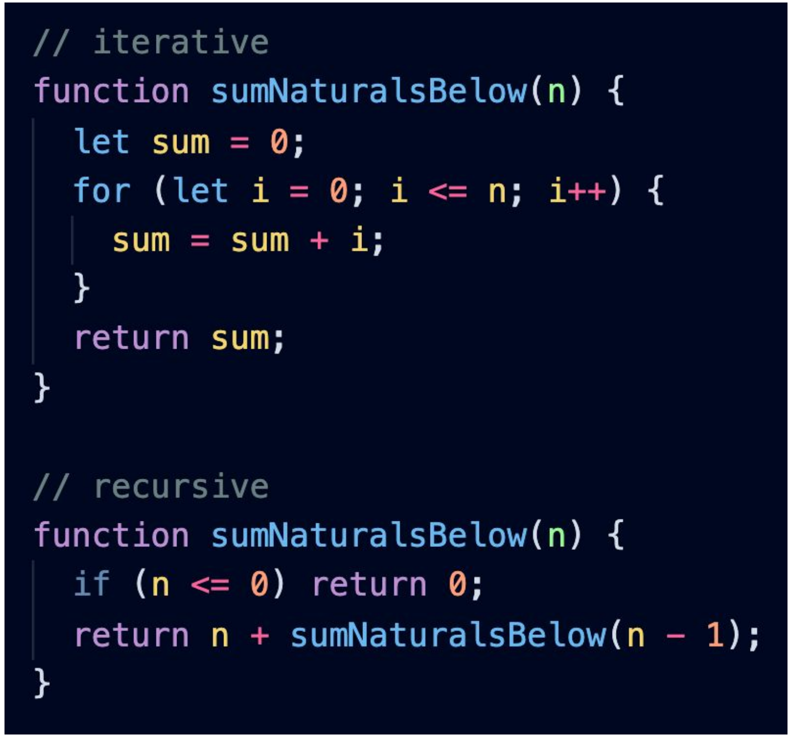
Recursion, like a for/while loop, requires terminating conditions



```
// Terminating function
function sumNaturalsBelow(n){
  if (n <= 0) return 0;
  return n + sumNaturalsBelow(n - 1);
}
```

Common Traits - Iterative/Recursive

- Initialized Value
- Changing Initial Value
- Terminating Condition

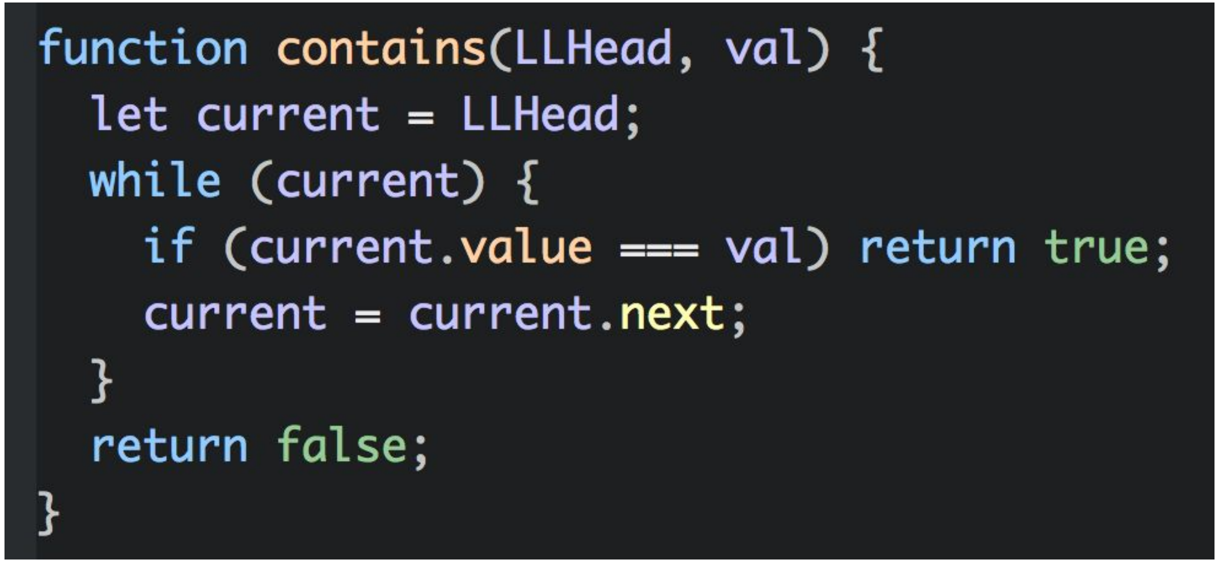


```
// iterative
function sumNaturalsBelow(n) {
  let sum = 0;
  for (let i = 0; i <= n; i++) {
    sum = sum + i;
  }
  return sum;
}

// recursive
function sumNaturalsBelow(n) {
  if (n <= 0) return 0;
  return n + sumNaturalsBelow(n - 1);
}
```

Why use recursion?

Recursive functions are frequently more readable, maintainable, and easier to understand

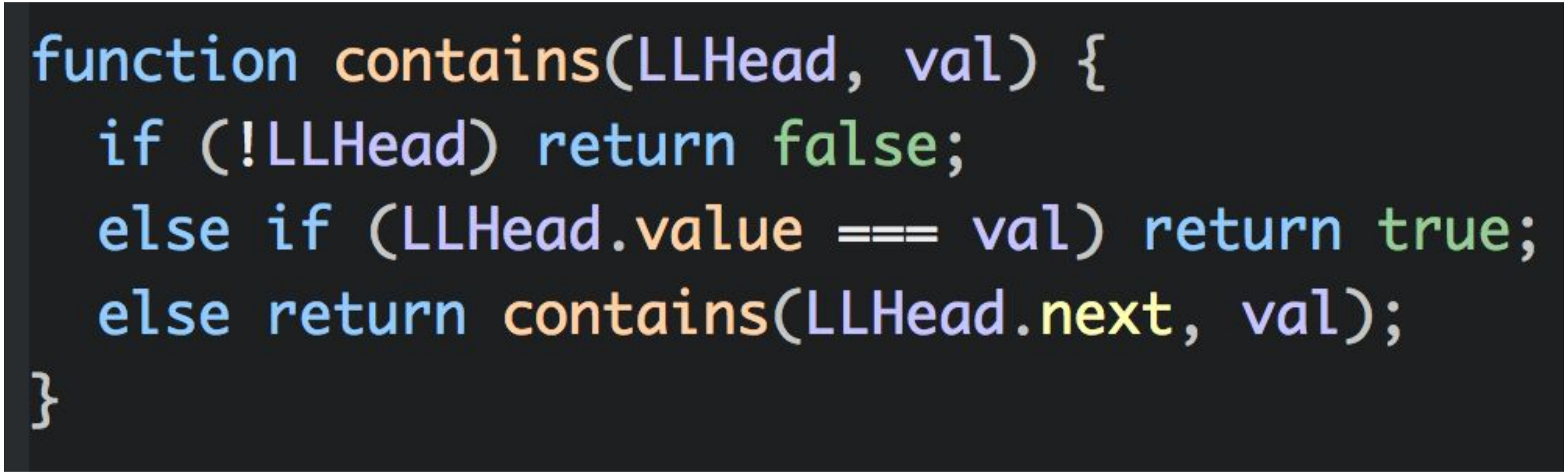
Without Recursion



```
function contains(LLHead, val) {
  let current = LLHead;
  while (current) {
    if (current.value === val) return true;
    current = current.next;
  }
  return false;
}
```

Recursive functions are frequently more readable, maintainable, and easier to understand

With Recursion



```
function contains(LLHead, val) {
  if (!LLHead) return false;
  else if (LLHead.value === val) return true;
  else return contains(LLHead.next, val);
}
```

Why are recursive algorithms easier to understand?

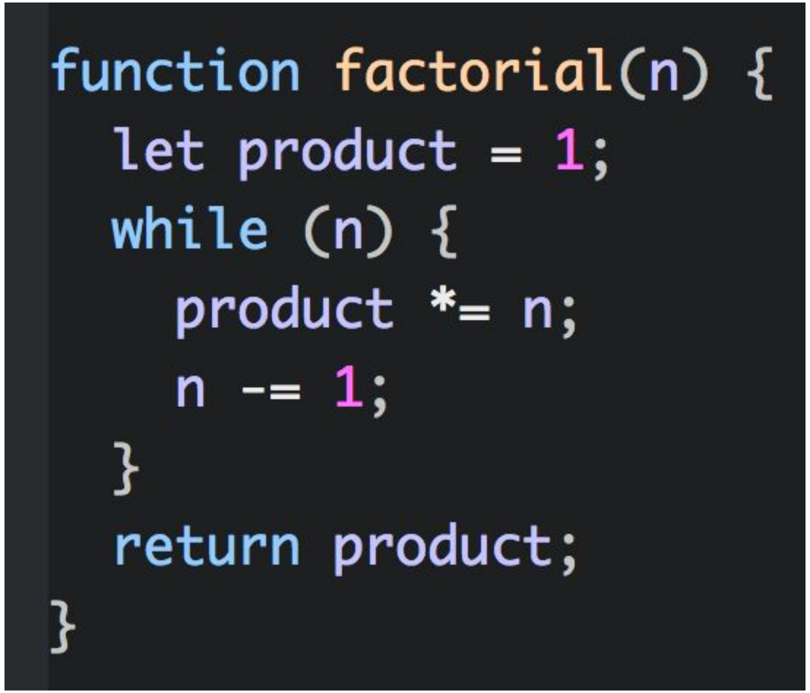
- Non-recursive functions tend to describe **how** to get to a solution.
- Recursive solutions describe **what** the solution is.
- Writing recursive functions often forces you to write **declarative** code instead of **imperative** code

Types of Recursion

Let's Compare Three ways to write a factorial algorithm

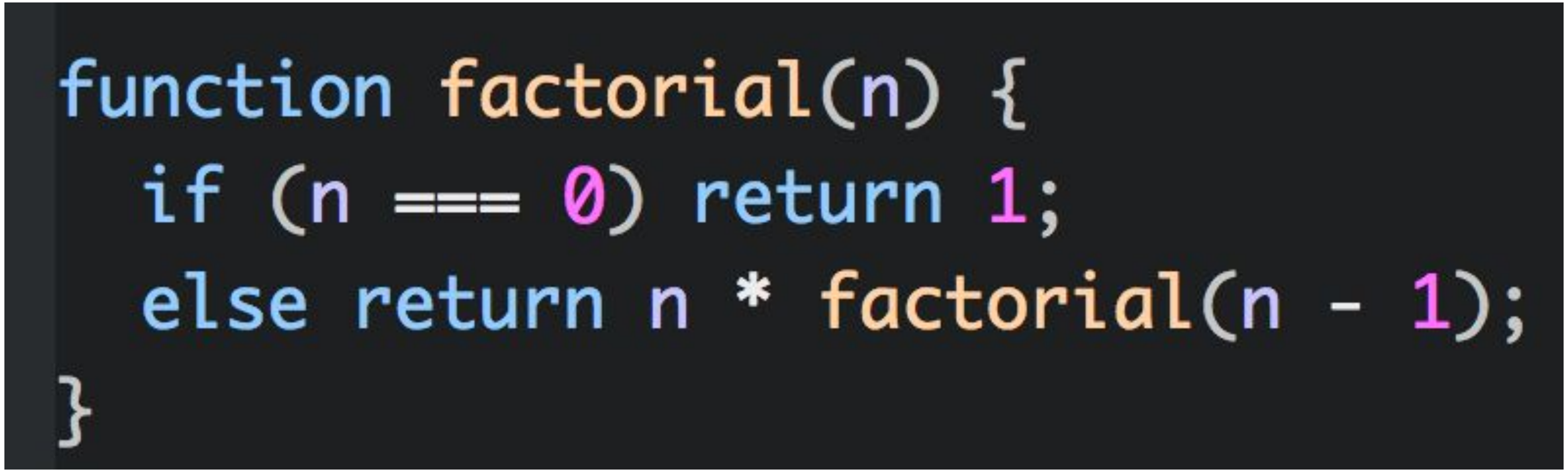
factorial(6) ==> 6 * 5 * 4 * 3 * 2 * 1

Factorial: Option #1



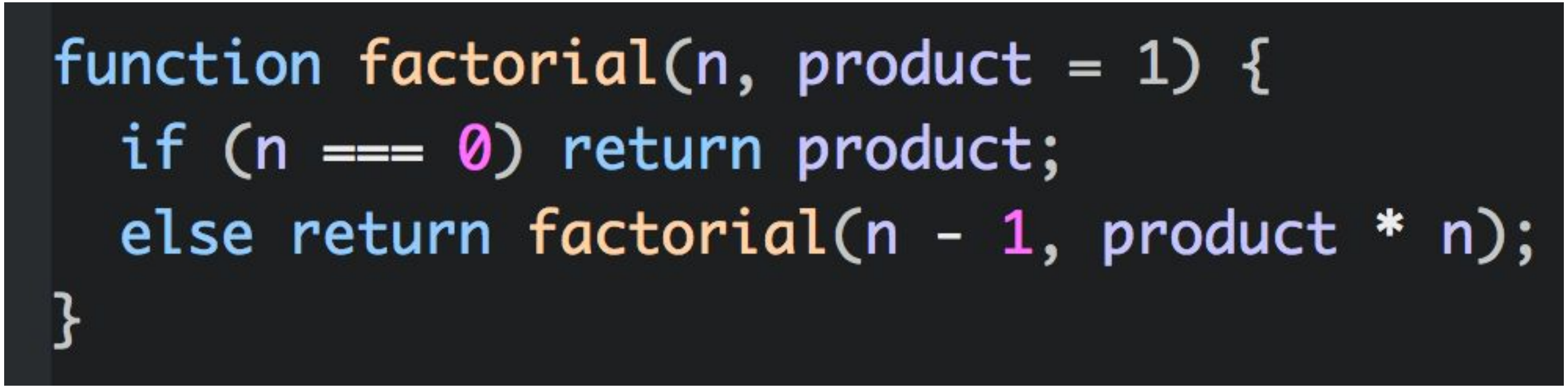
```
function factorial(n) {
  let product = 1;
  while (n) {
    product *= n;
    n -= 1;
  }
  return product;
}
```

Factorial: Option #2



```
function factorial(n) {
  if (n === 0) return 1;
  else return n * factorial(n - 1);
}
```

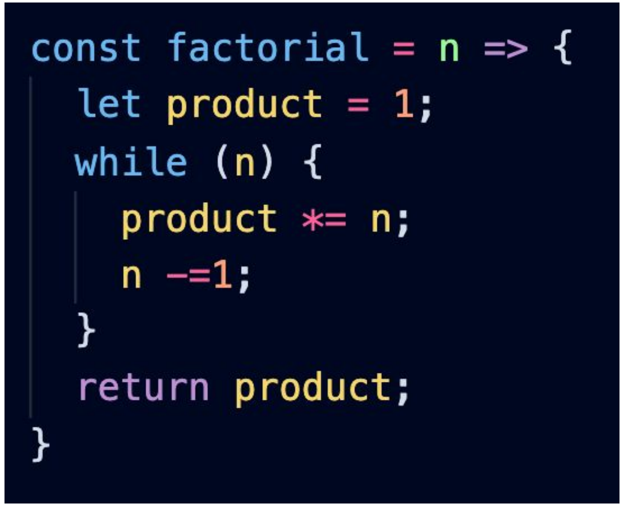
Factorial: Option #3



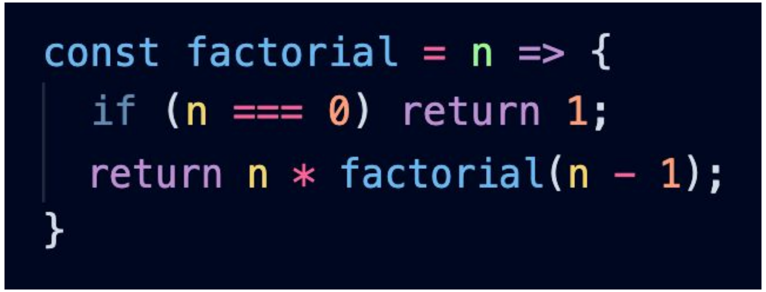
```
function factorial(n, product = 1) {
  if (n === 0) return product;
  else return factorial(n - 1, product * n);
}
```

Let’s Compare!

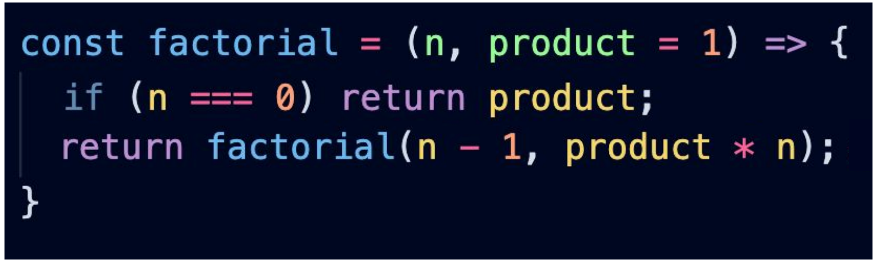
- *How is option #1 different from option #2 and #3?*



```
const factorial = n => {
  let product = 1;
  while (n) {
    product *= n;
    n -=1;
  }
  return product;
}
```



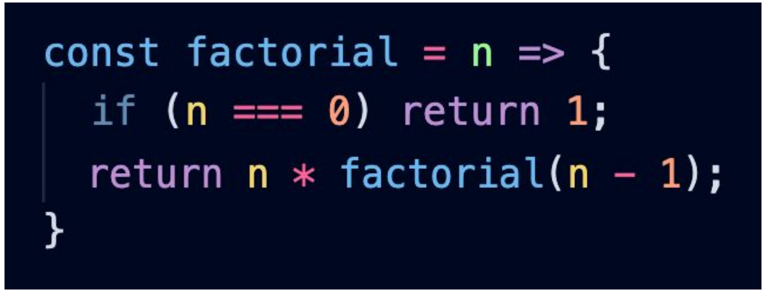
```
const factorial = n => {
  if (n === 0) return 1;
  return n * factorial(n - 1);
}
```



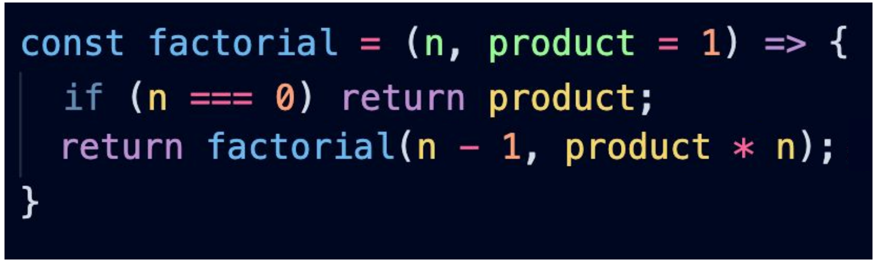
```
const factorial = (n, product = 1) => {
  if (n === 0) return product;
  return factorial(n - 1, product * n);
}
```

Let’s Compare!

- *How are option #2 and #3 similar? Different?*



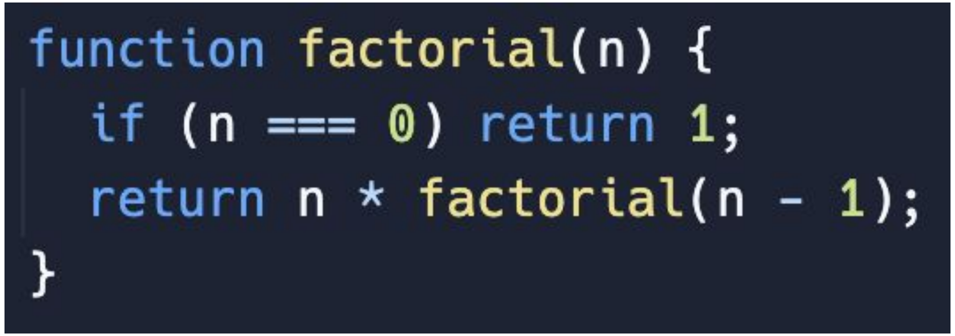
```
const factorial = n => {
  if (n === 0) return 1;
  return n * factorial(n - 1);
}
```



```
const factorial = (n, product = 1) => {
  if (n === 0) return product;
  return factorial(n - 1, product * n);
}
```

Linear Recursion:

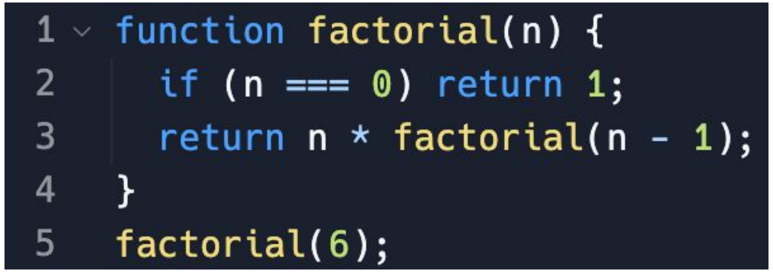
```
(factorial 6)
( * 6 (factorial 5) )
( * 6 ( * 5 (factorial 4) ) )
( * 6 ( * 5 ( * 4 (factorial 3) ) ) )
( * 6 ( * 5 ( * 4 ( * 3 (factorial 2) ) ) ) )
( * 6 ( * 5 ( * 4 ( * 3 ( * 2 (factorial 1) ) ) ) ) )
( * 6 ( * 5 ( * 4 ( * 3 ( * 2 ( * 1 (factorial 0) ) ) ) ) ) )
( * 6 ( * 5 ( * 4 ( * 3 ( * 2 ( * 1 * 1 ) ) ) ) ) )
( * 6 ( * 5 ( * 4 ( * 3 ( * 2 * 1 ) ) ) ) )
( * 6 ( * 5 ( * 4 ( * 3 * 2 ) ) ) )
( * 6 ( * 5 ( * 4 * 6 ) ) )
( * 6 ( * 5 * 24 ) )
( * 6 * 120 )
= 720
```



```
function factorial(n) {
  if (n === 0) return 1;
  return n * factorial(n - 1);
}
```

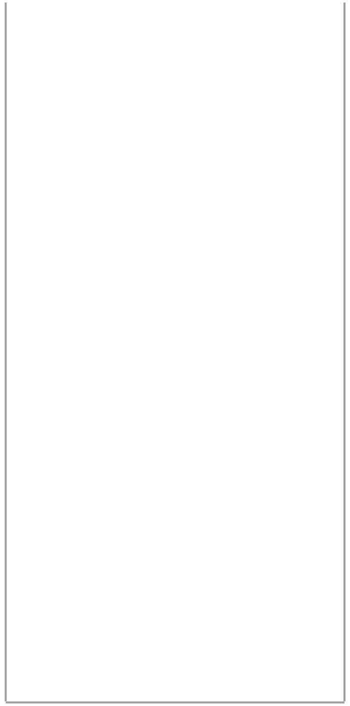
Time complexity: O(n)
Space complexity: O(n)

Linear Recursion:



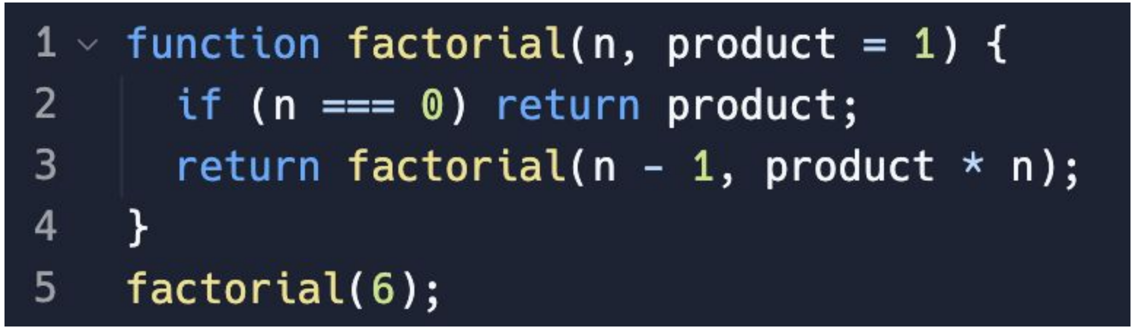
```
function factorial(n) {
  if (n === 0) return 1;
  return n * factorial(n - 1);
}
factorial(6);
```

Time complexity: O(n)
Space complexity: O(n)



Iterative Recursion (a.k.a. tail call recursion):

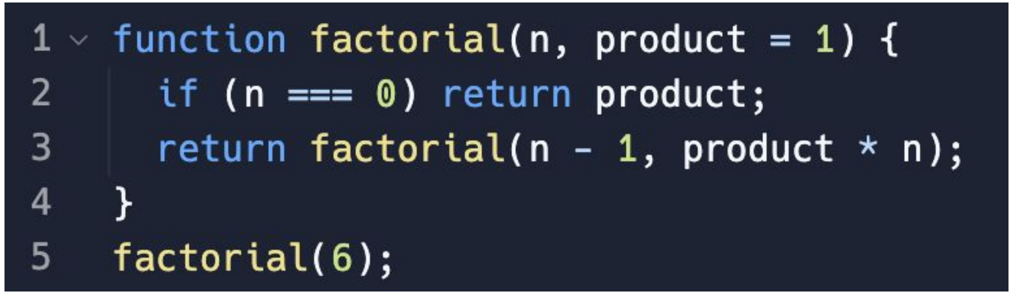
```
  (factorial 6)
⇒ (factorial 5, 6)
⇒ (factorial 4, 30)
⇒ (factorial 3, 120)
⇒ (factorial 2, 360)
⇒ (factorial 1, 720)
⇒ (factorial 0, 720)
⇒ 720
```



```javascript
function factorial(n, product = 1) {
  if (n === 0) return product;
  return factorial(n - 1, product * n);
}
factorial(6);
```

Time complexity: O(n)
Space complexity: O(1)

Iterative Recursion (a.k.a. tail call recursion):

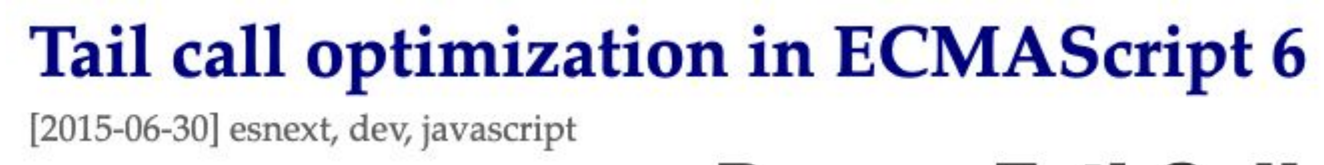


```
function factorial(n, product = 1) {
  if (n === 0) return product;
  return factorial(n - 1, product * n);
}
factorial(6);
```

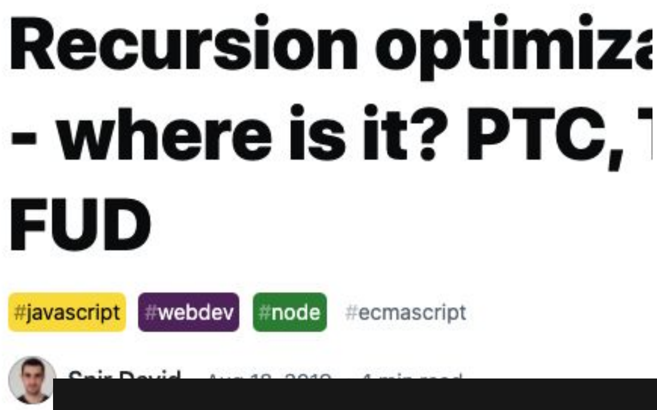
Time complexity: O(n)
Space complexity: O(1)

CALLSTACK

ES6 support for TCO

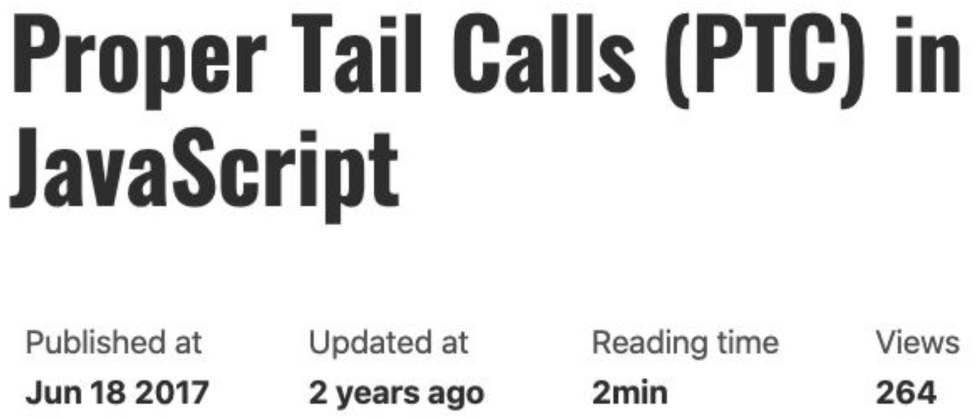

Tail call optimization in ECMAScript 6

[2015-06-30] esnext, dev, javascript

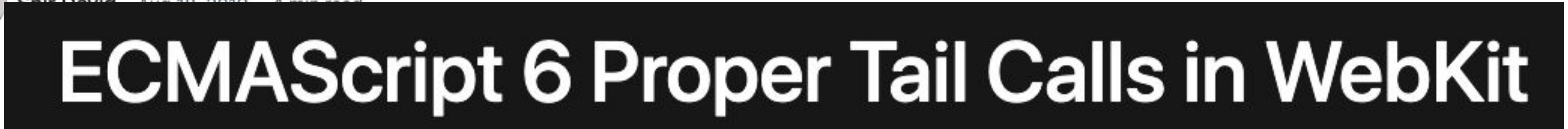

Recursion optimiza
- where is it? PTC, T
FUD

#javascript #webdev #node #ecmascript

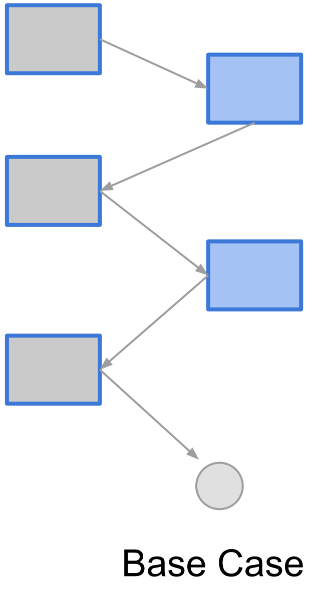
Snir David

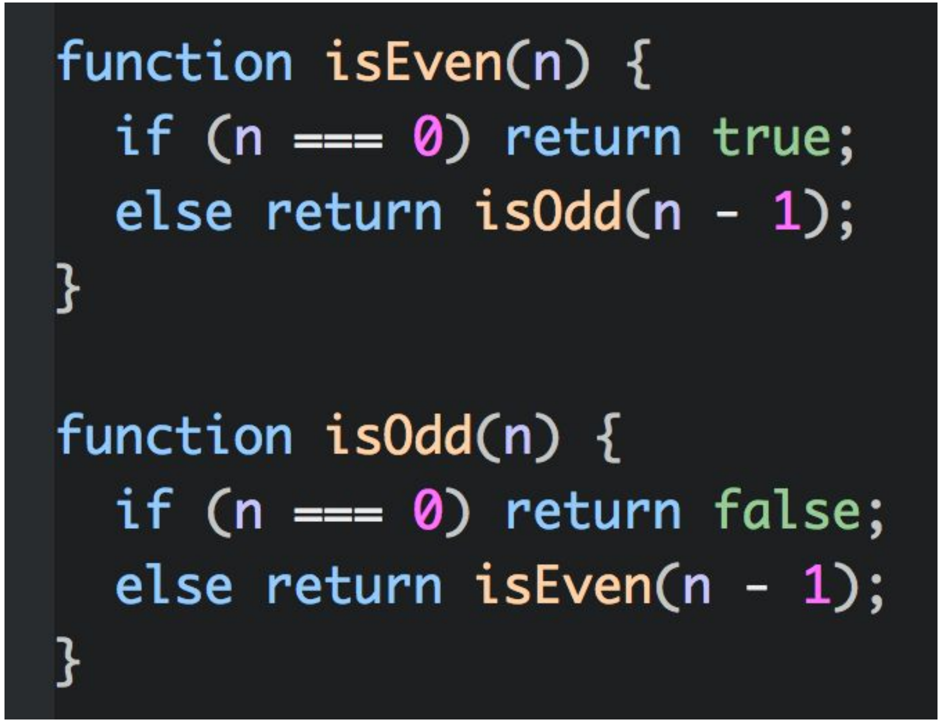

Proper Tail Calls (PTC) in JavaScript

Published at	Updated at	Reading time	Views
Jun 18 2017	2 years ago	2min	264


ECMAScript 6 Proper Tail Calls in WebKit

Other types of recursion: Mutual Recursion





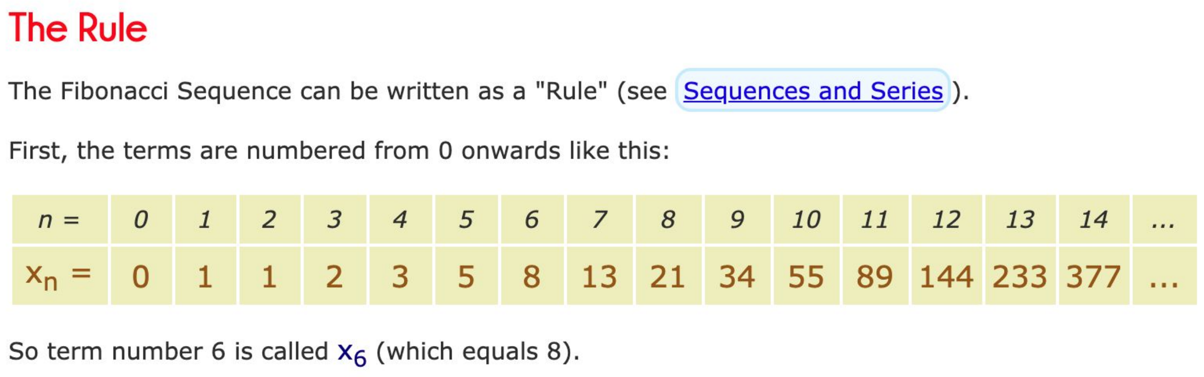
```
function isEven(n) {
  if (n === 0) return true;
  else return isOdd(n - 1);
}

function isOdd(n) {
  if (n === 0) return false;
  else return isEven(n - 1);
}
```

Other types of recursion: Mutual Recursion

JSON Parser!

Other types of recursion: Tree recursion (a.k.a multiple recursion)

Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34...



The Rule

The Fibonacci Sequence can be written as a "Rule" (see Sequences and Series).

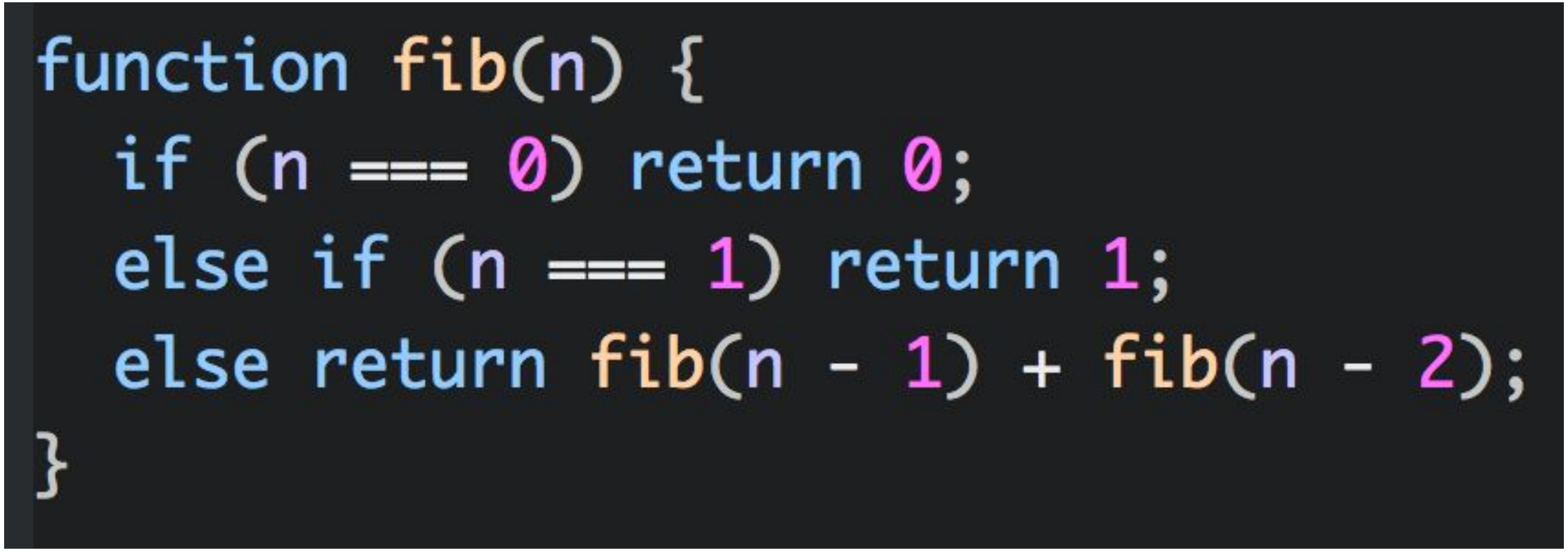
First, the terms are numbered from 0 onwards like this:

$n =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
$x_n =$	0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	...

So term number 6 is called x_6 (which equals 8).

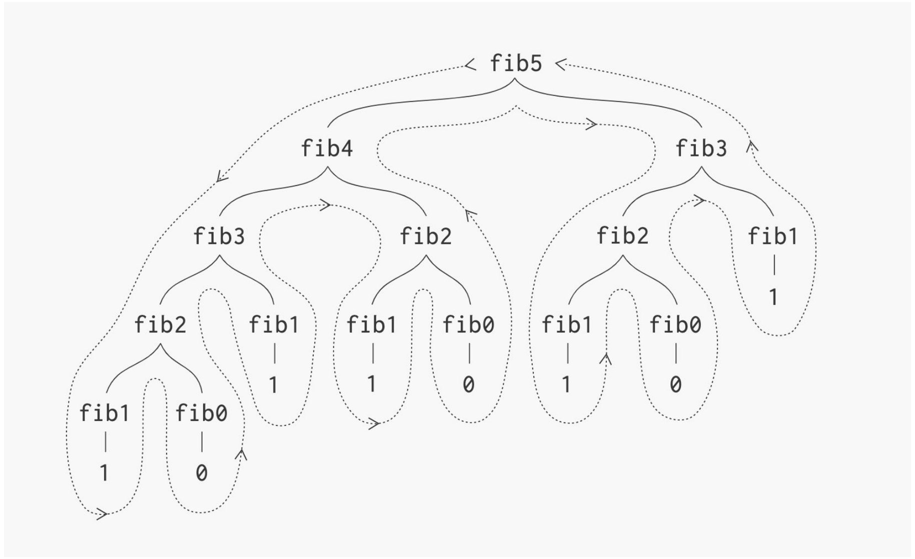
Other types of recursion: Tree recursion (a.k.a multiple recursion)

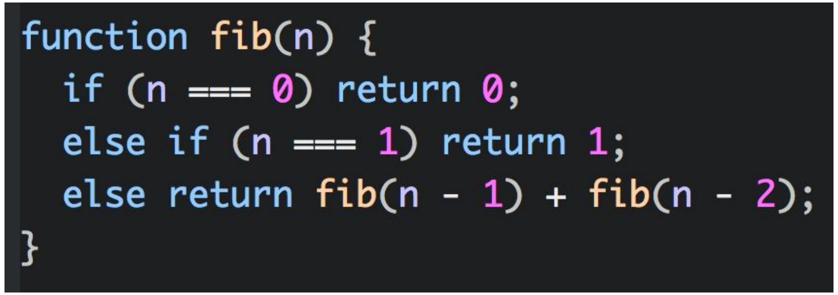
Finding the corresponding Fibonacci number at the nth term



```
function fib(n) {
  if (n === 0) return 0;
  else if (n === 1) return 1;
  else return fib(n - 1) + fib(n - 2);
}
```

Other types of recursion: Tree recursion (a.k.a multiple recursion)





```
function fib(n) {
  if (n === 0) return 0;
  else if (n === 1) return 1;
  else return fib(n - 1) + fib(n - 2);
}
```

Finding the corresponding Fibonacci number at the nth term

Summary

- A function is recursive if it calls itself in its own definition.
- Recursion allows us to write **declarative** code, rather than **imperative** code.
- Not all browsers implement Tail Call Optimization (TCO)
- Types of recursion:
  - Linear recursion
  - Tail recursion
  - Mutual recursion
  - Tree recursion

Further Reading

- Recursion in functional JavaScript (SitePoint)
- Recursion (JavaScript) (Microsoft)