Call Stacks and Execution Frames

7 min

A recursive approach requires the function invoking itself **with different arguments.** How does the computer keep track of the various arguments and different function invocations if it's the same function definition?

Repeatedly invoking functions may be familiar when it occurs sequentially, but it can be jarring to see this invocation occur **within a function definition**.

Languages make this possible with *call stacks* and *execution contexts*.

Stacks, a data structure, follow a strict

Preview: Docs Loading link description

protocol

for the order data enters and exits the structure: **the last thing to enter is the first thing to leave**.

Your programming language often manages the call stack, which exists outside of any specific function. This call stack tracks the ordering of the different function invocations, so the **last function to enter the call stack is the first function to exit the call stack**.

We can think of execution contexts as the specific values we plug into a function call.

A function which adds two numbers:

Invoking the function with 3 and 4 as arguments...
execution context:
X = 3
Y = 4

Invoking the function with 6 and 2 as arguments...
execution context:
X = 6
Y = 2

Consider a pseudo-code function which sums the integers in an array:

```
function, sum_list
  if list has a single element
    return that single element
  otherwise...
    add first element to value of sum_list called with every element minus the first
```

This function will be invoked as many times as there are elements within the list! Let's step through:

CALL STACK EMPTY

____

Our first function call...
sum_list([5, 6, 7])

CALL STACK CONTAINS

____
sum_list([5, 6, 7])
with the execution context of a list being [5, 6, 7]

____

Base case, a list of one element not met.
We invoke sum_list with the list of [6, 7]...

CALL STACK CONTAINS

____
sum_list([6, 7])
with the execution context of a list being [6, 7]

____
sum_list([5, 6, 7])
with the execution context of a list being [5, 6, 7]

____

Base case, a list of one element not met.
We invoke sum_list with the list of [7]...

CALL STACK CONTAINS

____
sum_list([7])
with the execution context of a list being [7]

____
sum_list([6, 7])
with the execution context of a list being [6, 7]

____
sum_list([5, 6, 7])
with the execution context of a list being [5, 6, 7]

____

We've reached our base case! List is one element.
We return that one element.
This return value does two things:

1) "pops" sum_list([7]) from CALL STACK.
2) provides a return value for sum_list([6, 7])

CALL STACK CONTAINS

____
sum_list([6, 7])
with the execution context of a list being [6, 7]
RETURN VALUE = 7

____
sum_list([5, 6, 7])
with the execution context of a list being [5, 6, 7]

____

sum_list([6, 7]) waits for the return value of sum_list([7]), which it just received.

sum_list([6, 7]) has resolved and "popped" from the call stack...

CALL STACK contains

____

sum_list([5, 6, 7])
with the execution context of a list being [5, 6, 7]
RETURN VALUE = 6 + 7

____

sum_list([5, 6, 7]) waits for the return value of sum_list([6, 7]), which it just received.
sum_list([5, 6, 7]) has resolved and "popped" from the call stack.

CALL STACK is empty

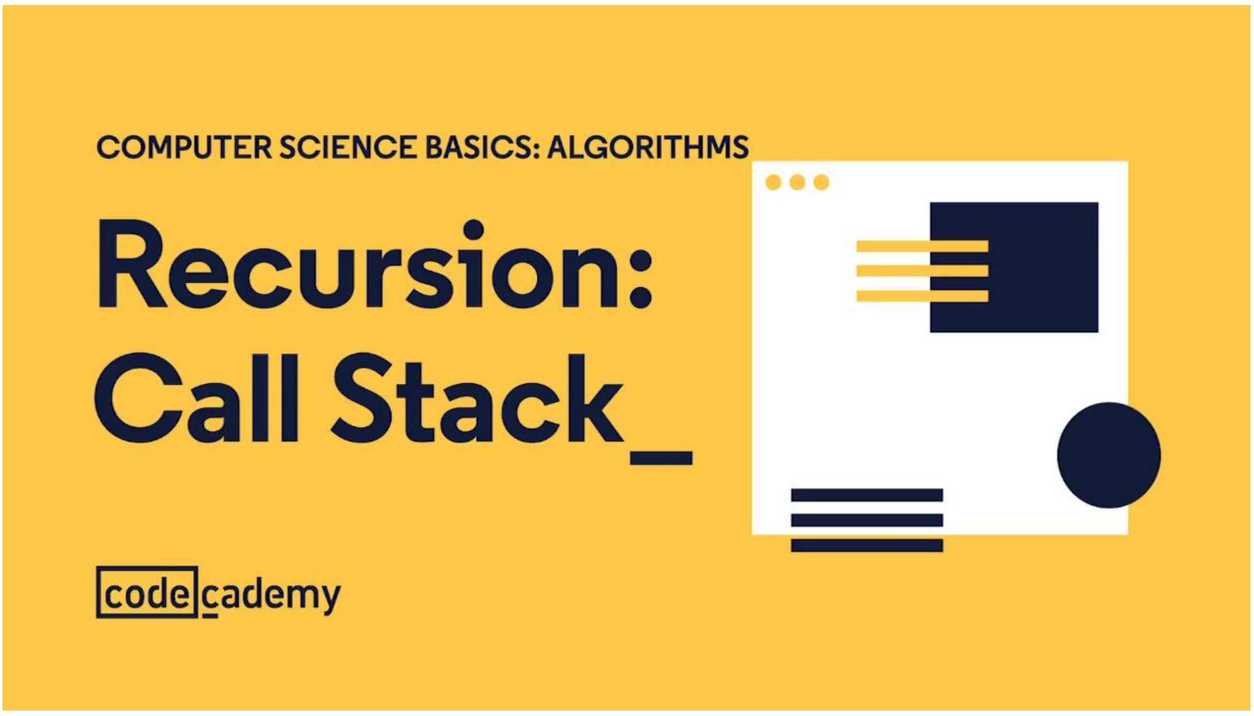
____

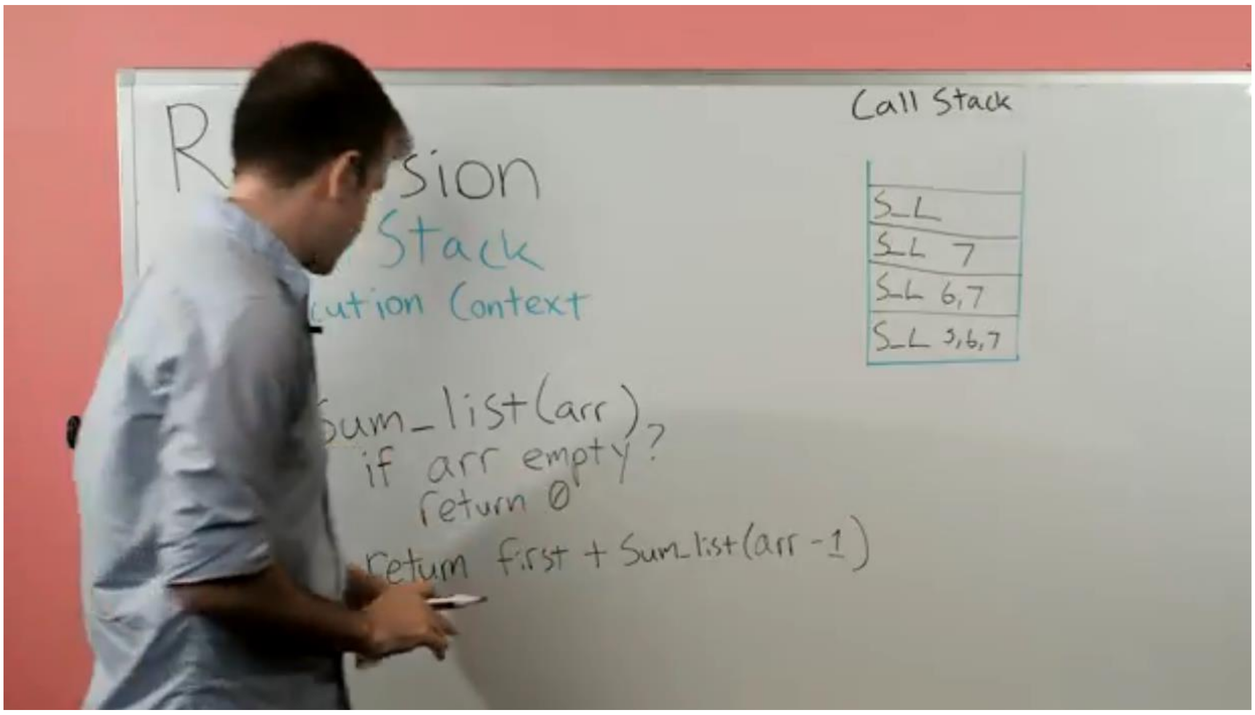
RETURN VALUE = (5 + 6 + 7) = 18

Instructions

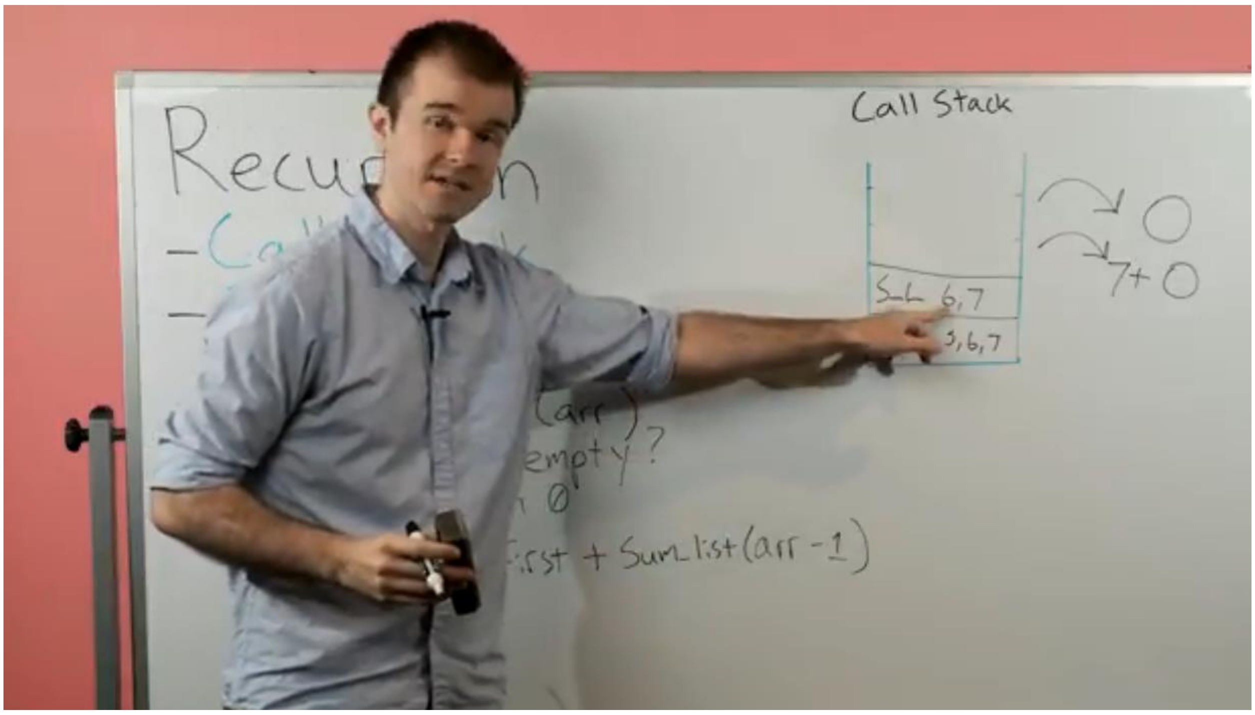
When a recursive function enters the base case without any recursive calls, will the call stack be empty?

Why or why not?

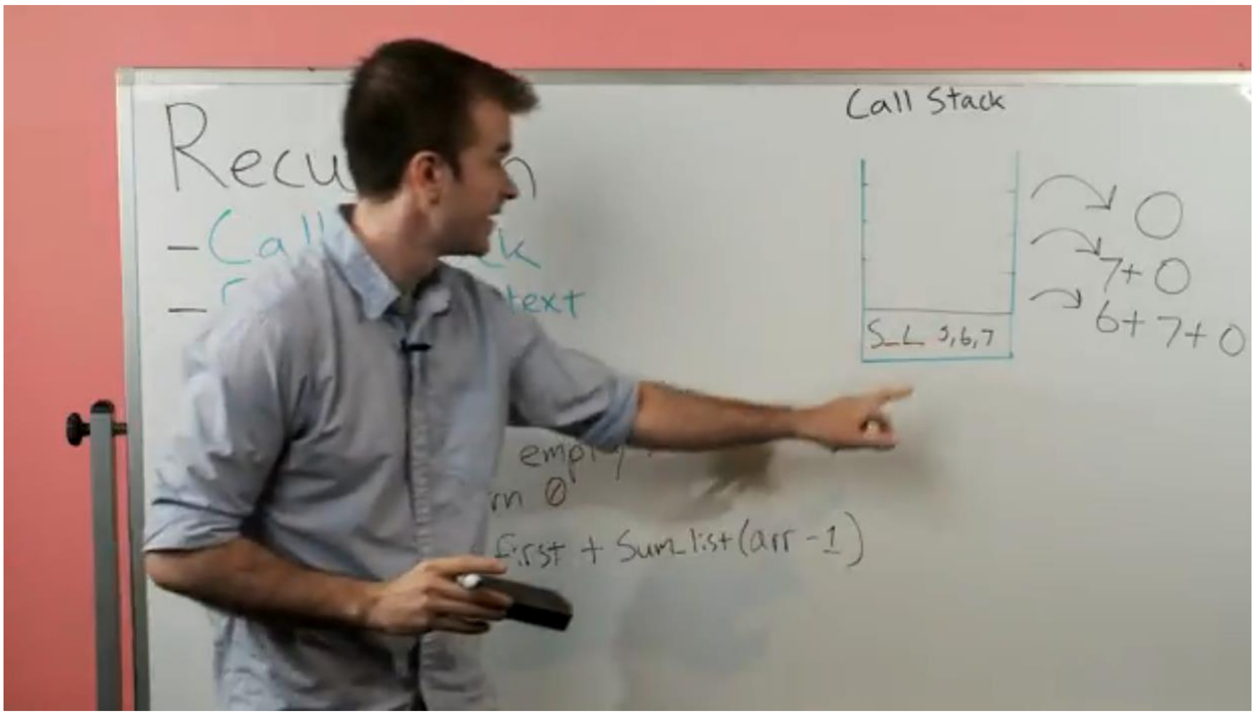


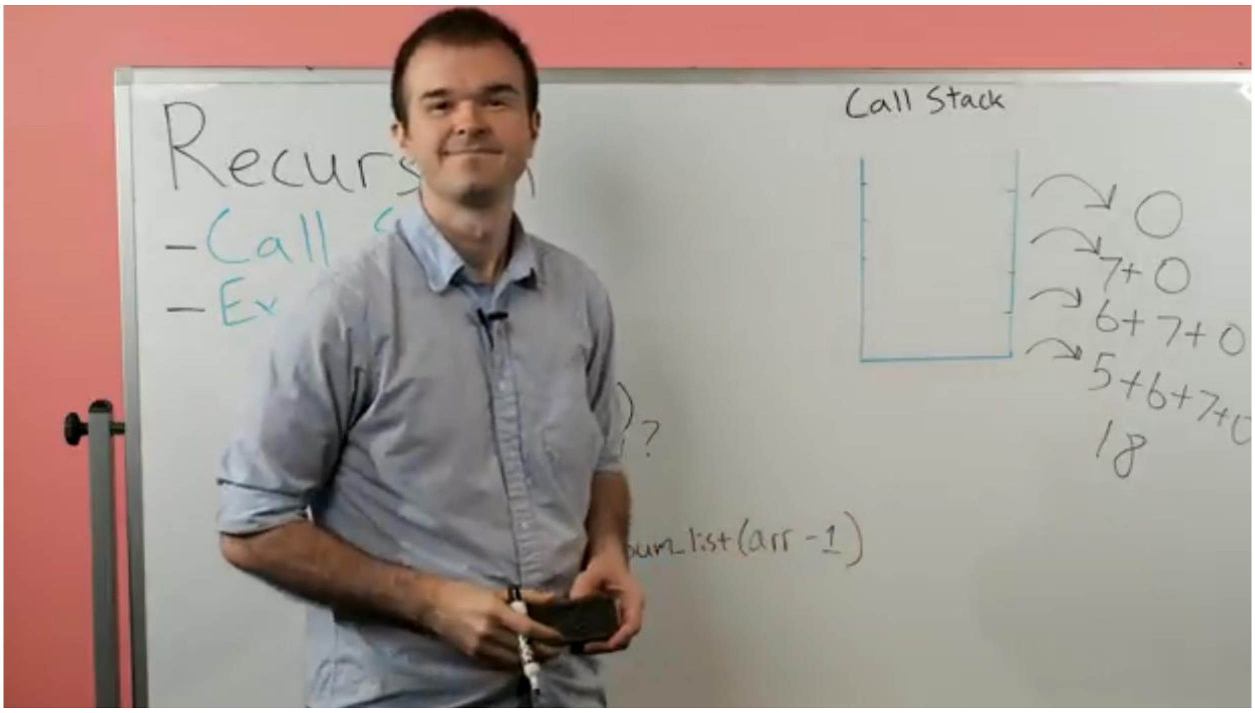


Call Stack
S_L
S_L 7
S_L 6,7
S_L 5,6,7
R sion
Stack
cution Context
Sum_list(arr)
if arr empty?
return 0
return first + Sum_list(arr - 1)



Call Stack
Recu n
0
7+0
S_L 6,7
5,6,7
(arr)
empty?
0
first + Sum_list(arr - 1)


Call Stack
S_L 5,6,7
0
7+0
6+7+0
first + Sum_list(arr -1)


Call Stack
0
7+0
6+7+0
5+6+7+0
18
sum_list(arr -1)