Recursive Function Call Tracing

Supplement to Recursion lecture

Tracing the Function Calls

Start with the original function call.

```
def addCards(cards):
    if cards == [ ]:
        return 0
    else:
        smallerProblem = cards[1:]
        smallerResult = addCards(smallerProblem)
        return cards[0] + smallerResult

addCards([5, 2, 7, 3])
```

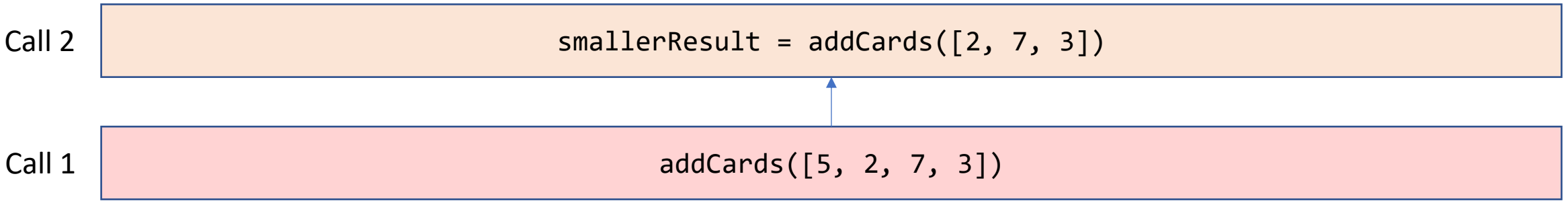
Call 1 | `addCards([5, 2, 7, 3])`

Tracing the Function Calls

We go to the recursive case and set up a local variable `smallerProblem`. Then call `addCards` again on that variable, making another function call.

```
def addCards(cards):
    if cards == [ ]:
        return 0
    else:
        smallerProblem = cards[1:]
        smallerResult = addCards(smallerProblem)
        return cards[0] + smallerResult

addCards([5, 2, 7, 3])
```

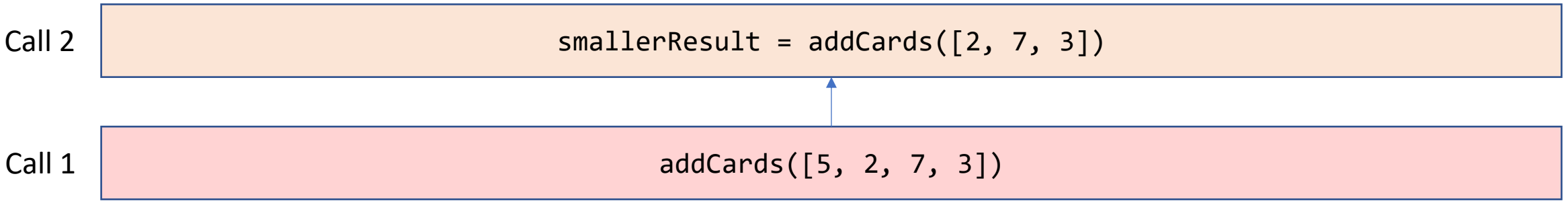


Tracing the Function Calls

When we run through `addCards` a second time, there's a **new local state**. `cards` is now `[2, 7, 3]`. `smallerProblem` is now `[7, 3]`.

```
def addCards(cards):
    if cards == [ ]:
        return 0
    else:
        smallerProblem = cards[1:]
        smallerResult = addCards(smallerProblem)
        return cards[0] + smallerResult

addCards([5, 2, 7, 3])
```

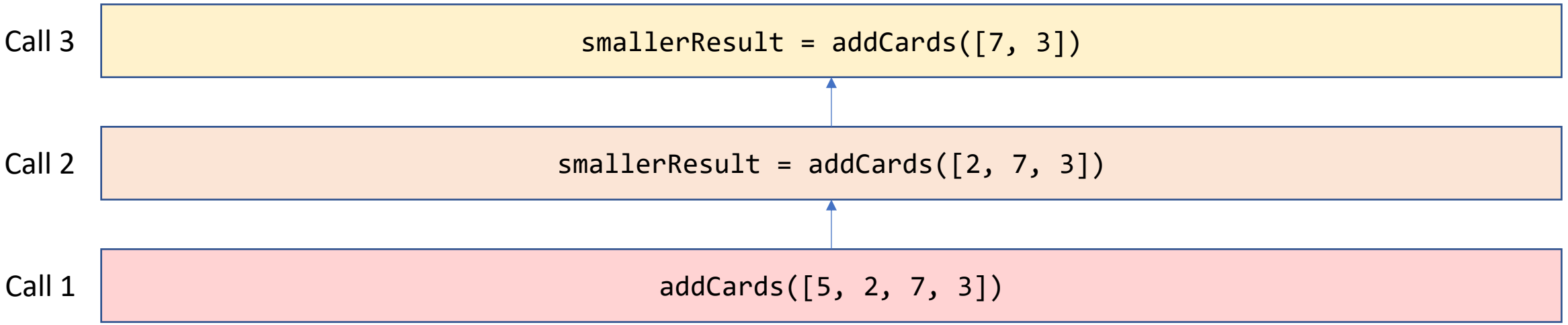


Tracing the Function Calls

Call `addCards` again, this time on `[7, 3]`. Note that the function call tracing helps us keep track of **all** previous calls.

```
def addCards(cards):
    if cards == [ ]:
        return 0
    else:
        smallerProblem = cards[1:]
        smallerResult = addCards(smallerProblem)
        return cards[0] + smallerResult

addCards([5, 2, 7, 3])
```

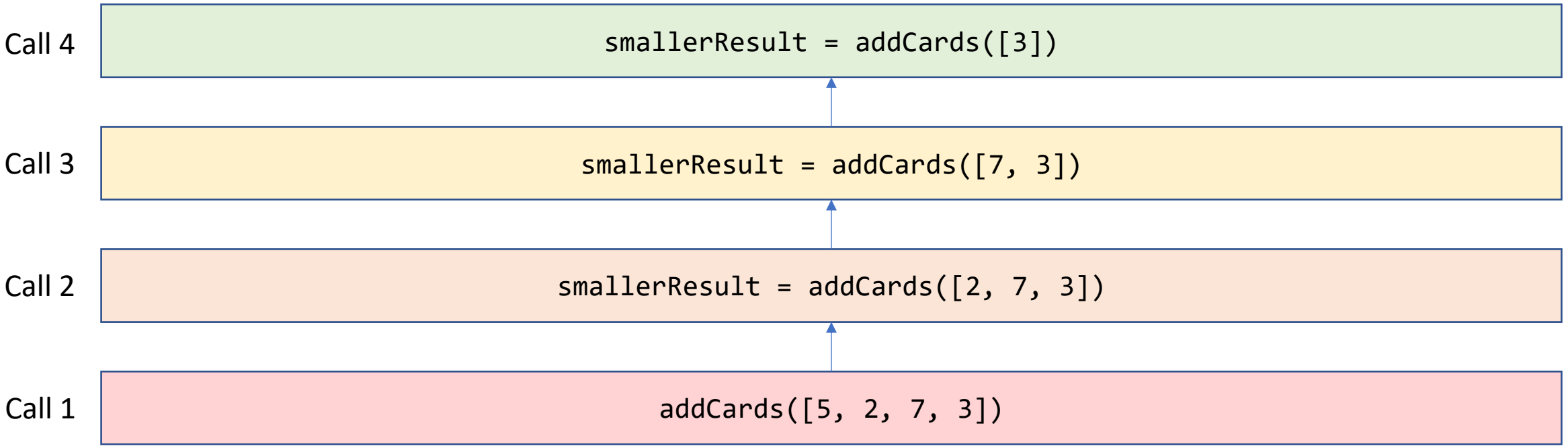


Tracing the Function Calls

Now we run the function with `cards` set to `[7, 3]`. `smallerProblem` becomes `[3]`; we call the function again.

```
def addCards(cards):
    if cards == [ ]:
        return 0
    else:
        smallerProblem = cards[1:]
        smallerResult = addCards(smallerProblem)
        return cards[0] + smallerResult

addCards([5, 2, 7, 3])
```

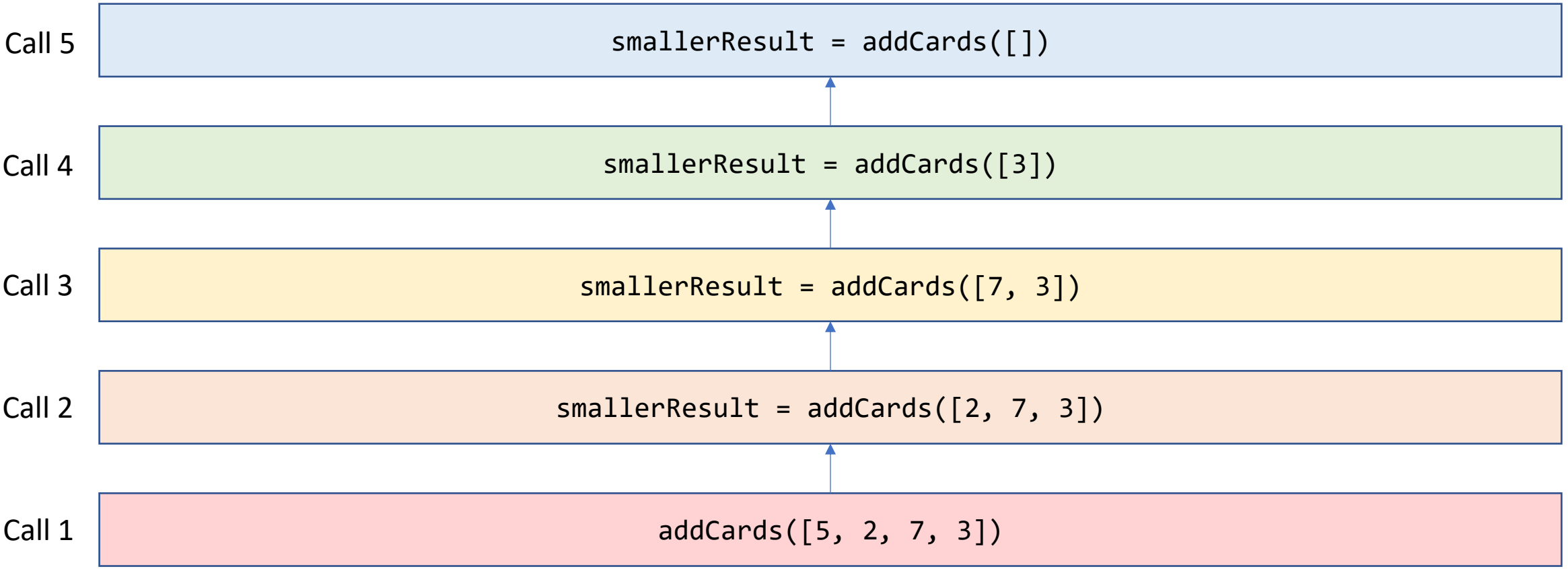


Tracing the Function Calls

Run the function with `cards` set to `[3]`. `smallerProblem` becomes `[]`; run the function again.

```
def addCards(cards):
    if cards == [ ]:
        return 0
    else:
        smallerProblem = cards[1:]
        smallerResult = addCards(smallerProblem)
        return cards[0] + smallerResult

addCards([5, 2, 7, 3])
```

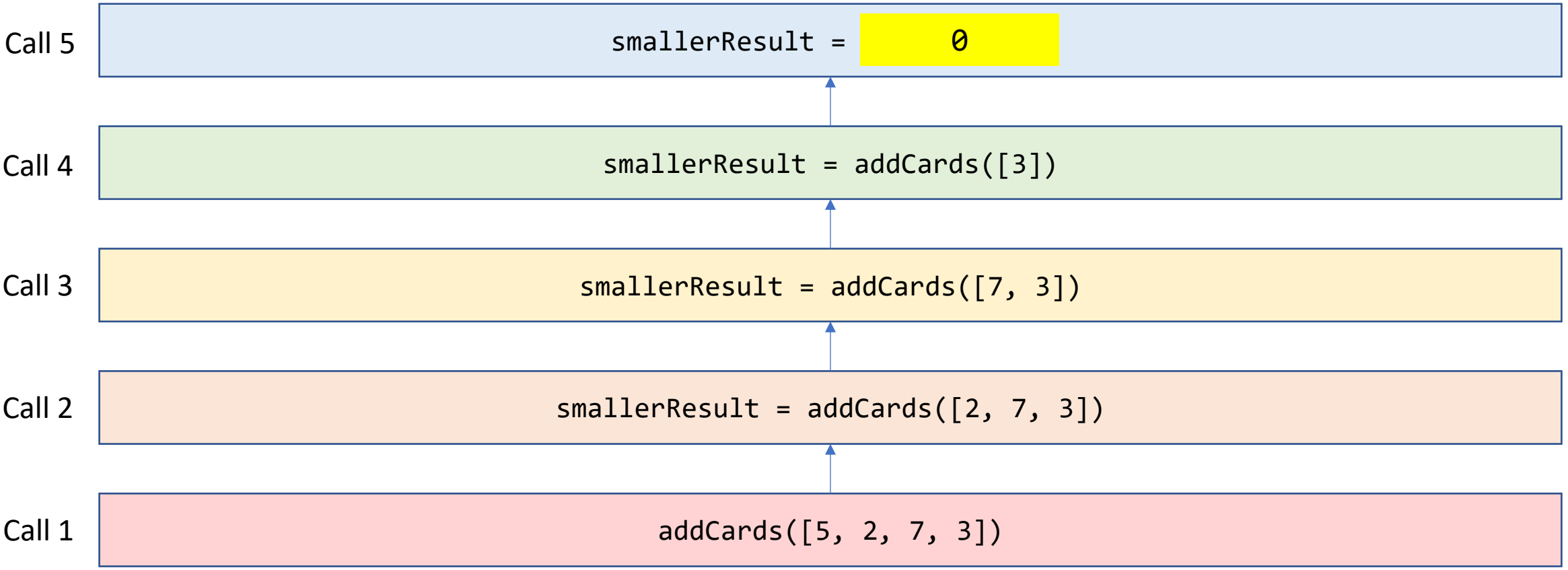


Tracing the Function Calls

Now we finally reach the base case. `addCards([])` returns `0` immediately, so `0` takes the place of the function call in the previous bookmark (in Call #5).

```
def addCards(cards):
    if cards == [ ]:
        return 0
    else:
        smallerProblem = cards[1:]
        smallerResult = addCards(smallerProblem)
        return cards[0] + smallerResult

addCards([5, 2, 7, 3])
```

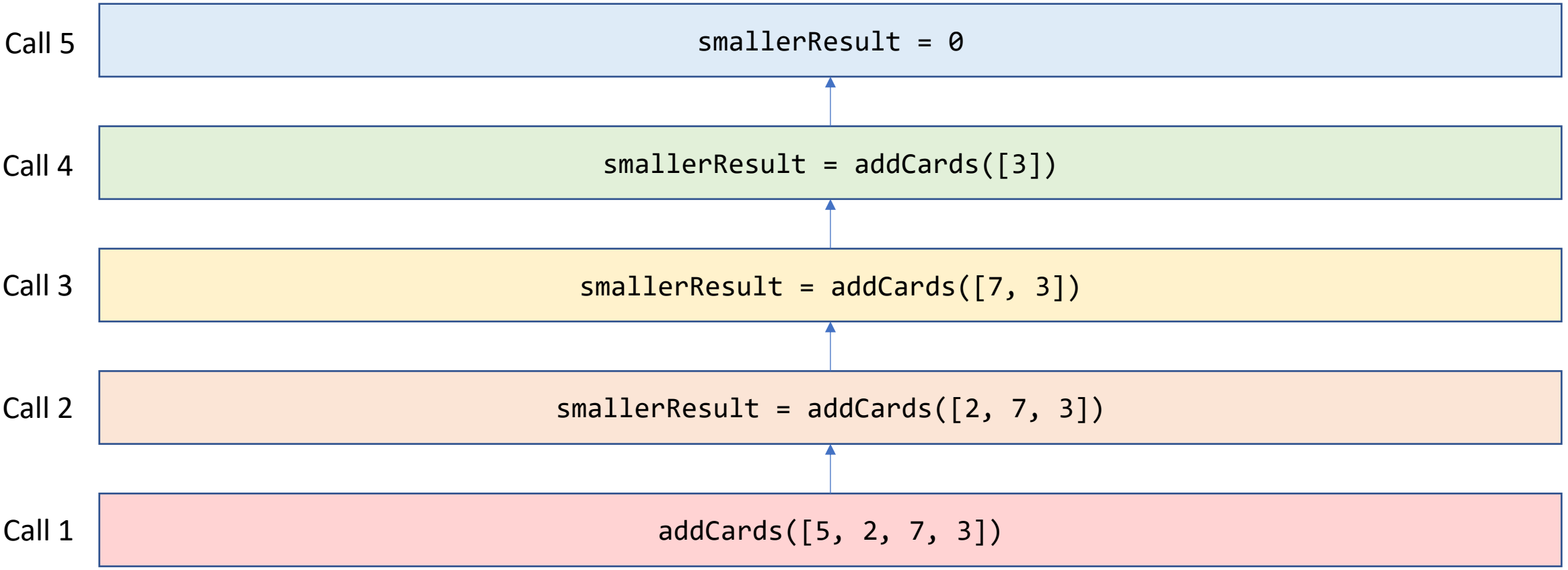


Tracing the Function Calls

What does Python do next? It remembers the **local state** of each call! On the blue level, it knows that `cards` is `[3]` and `smallerResult` is now `0`.

```
def addCards(cards):
    if cards == [ ]:
        return 0
    else:
        smallerProblem = cards[1:]
        smallerResult = addCards(smallerProblem)
        return cards[0] + smallerResult

addCards([5, 2, 7, 3])
```

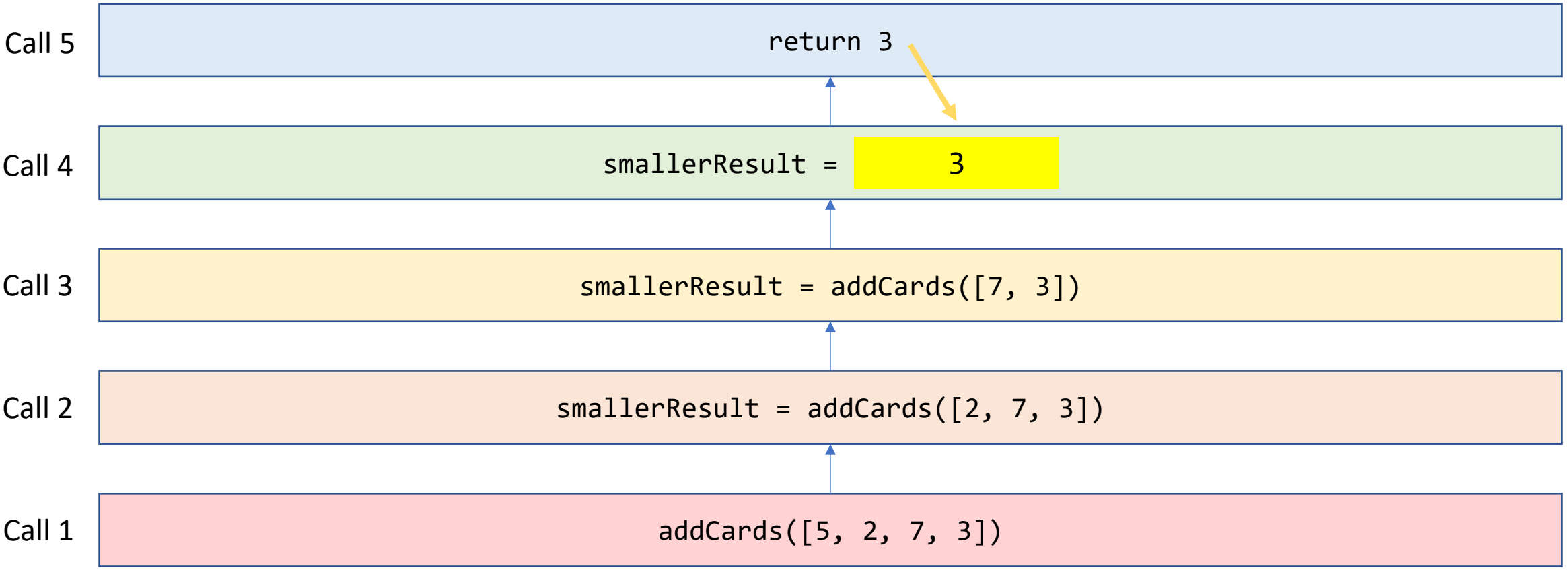


Tracing the Function Calls

Add `3 + 0` to get `3`; this can be sent back as the returned value to the previous level of the function calls (in Call #4).

```
def addCards(cards):
    if cards == [ ]:
        return 0
    else:
        smallerProblem = cards[1:]
        smallerResult = addCards(smallerProblem)
        return cards[0] + smallerResult

addCards([5, 2, 7, 3])
```

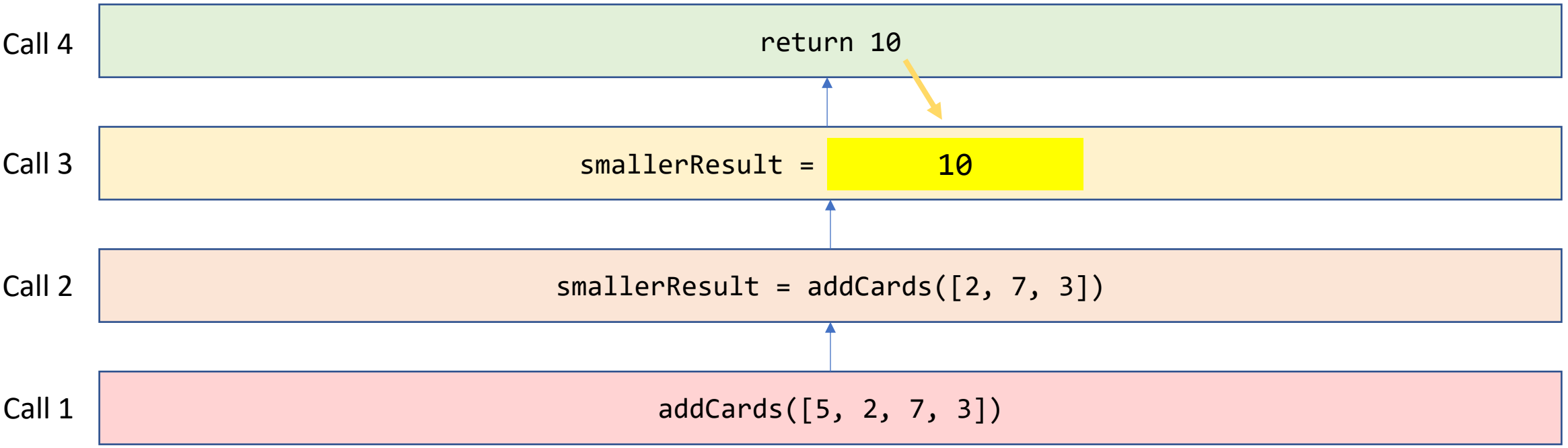


Tracing the Function Calls

At this level, `cards` is `[7, 3]` and `smallerResult` is `3`. `7 + 3` gives us a returned value of `10`.

```
def addCards(cards):
    if cards == [ ]:
        return 0
    else:
        smallerProblem = cards[1:]
        smallerResult = addCards(smallerProblem)
        return cards[0] + smallerResult

addCards([5, 2, 7, 3])
```

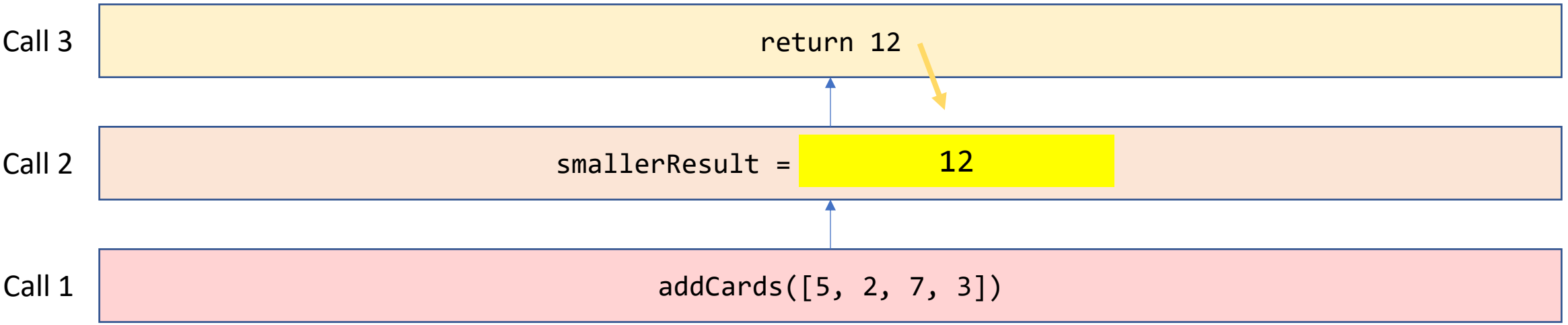


Tracing the Function Calls

At the next level, `cards` is `[2, 7, 3]` and `smallerResult` is `10`. `2 + 10` gives us a returned value of `12`.

```
def addCards(cards):
    if cards == [ ]:
        return 0
    else:
        smallerProblem = cards[1:]
        smallerResult = addCards(smallerProblem)
        return cards[0] + smallerResult

addCards([5, 2, 7, 3])
```

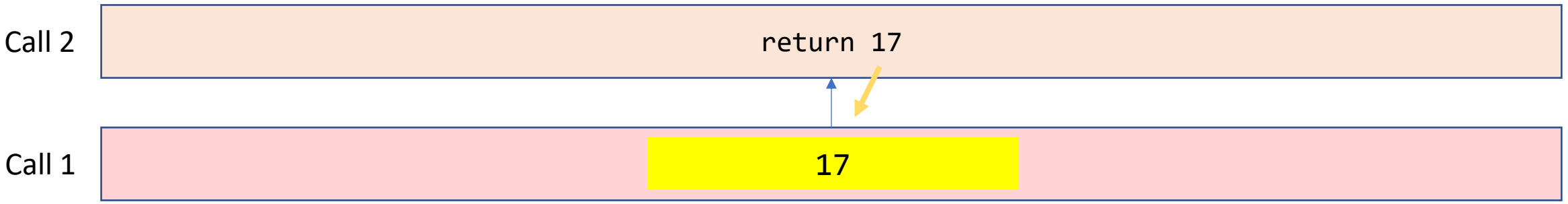


Tracing the Function Calls

Now `cards` is `[5, 2, 7, 3]` and `smallerResult` is `12`. Add `5 + 12` to get a returned value of `17`.

```
def addCards(cards):
    if cards == [ ]:
        return 0
    else:
        smallerProblem = cards[1:]
        smallerResult = addCards(smallerProblem)
        return cards[0] + smallerResult

addCards([5, 2, 7, 3])
```



Tracing the Function Calls

We've finally reached the original call! It evaluates to 17, and we've finished calculating the final answer.

```
def addCards(cards):
    if cards == [ ]:
        return 0
    else:
        smallerProblem = cards[1:]
        smallerResult = addCards(smallerProblem)
        return cards[0] + smallerResult

addCards([5, 2, 7, 3])
```

Call 1

17