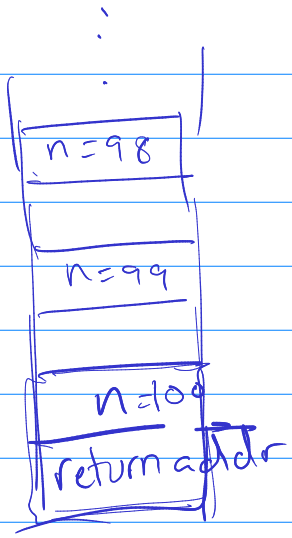

n=98
n=99
n=100
return addr

Stack size = 1 GB

$= 1024 \times 1024 \times 1024$ bytes

Each call to sum must remember

- n (8 bytes)
- return address (8 bytes)

Notice that $\frac{1024 \times 1024 \times 1024}{33570783 \times (8+8) \times 2} \approx 1$

33570783: Largest input n that we can supply

2: Unexplained.

```
sum n = if n=0 then 0
        else n + sum(n-1)
```

```
int ans = 0
for (i:0 ; i≤n; i++)
    ans = ans+i
```

---

sum 5 $\Rightarrow$ 5 + sum 4

$\Rightarrow$ 5 + (4 + sum 3)

$\overset{*}{\Rightarrow}$ 5 + (4 + (3 + (2 + (1 + 0))))

ans = 0

ans = ans + 1 = 1

ans = ans + 2 = 3

ans = ans + 3 = 6

ans = ans + 4 = 10

(((((0+1)+2)+3)+4)+5

C

```
int ans = 0
for (i=0; i<=n; i++)
    ans = ans+i
```

OCaml

```
helper ans n
  = if n=0 then ans
    else helper (ans+n) (n-1)
```

Before fn. application

Before fn. application

The fn does not need to do anything "after" the recursive call.

Empty continuation

```
sum n = n + sum (n-1)
```

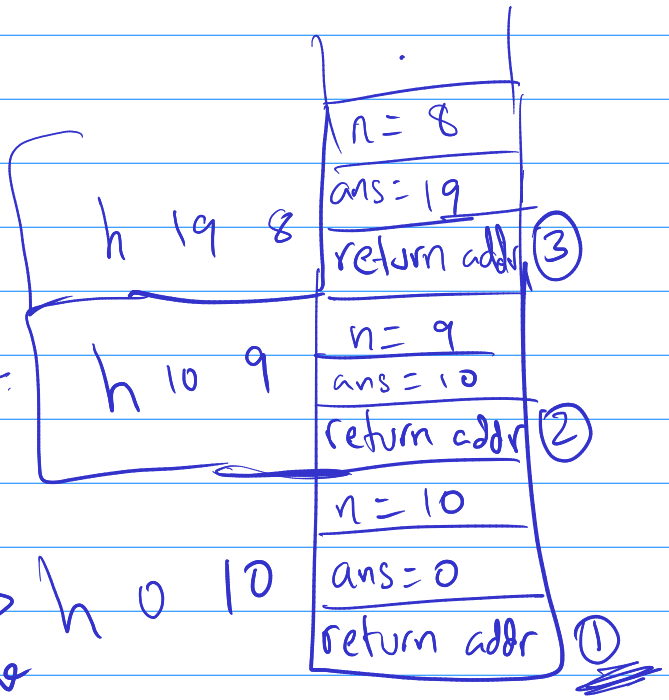
Recursive call

Addition happens "afterwards"

Continuation

helper 0 10

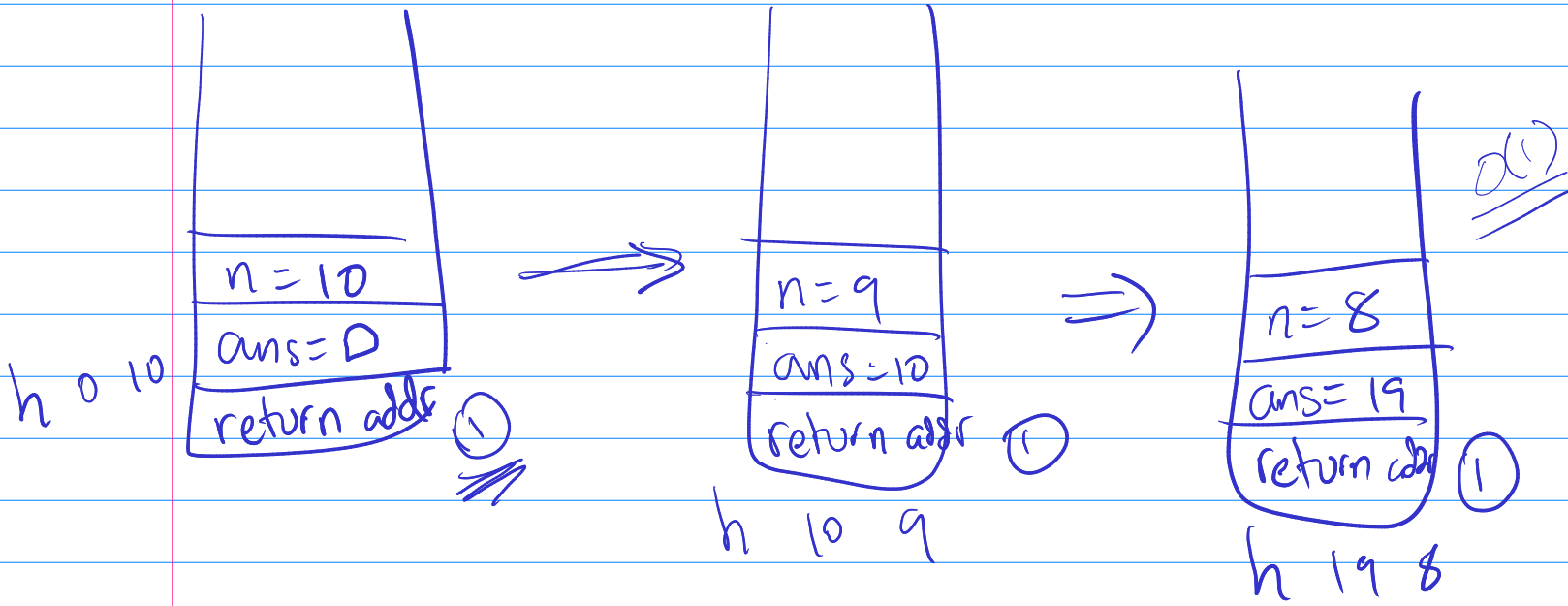
Given that we have an empty continuation



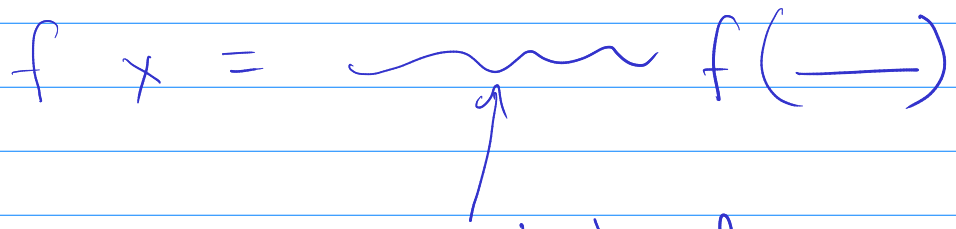
O(n)

Whatever value this fn returns will also be returned by this outside fn

---



O(n)



Work to be done after the recursive call

"Continuation".

Optimization: If continuation is empty then don't allocate a new stack frame.

Reuse

Tail call optimization

Q: Why no tail call optimization in ~~GCC~~ JavaScript?

A1: Maybe C is too complicated?

A2: May GCC developers are stupid?

A3: Maybe they have good reasons not to?

---

tail call tail recursion.

my rev [1 2 3 4 5]

(my rev [2 3 4 5]) @ [1]

---

my rev [1 2 3 4 5]

↓ tl / pattern match

TODO list

~~my rev~~ [2 3 4 5] [1]

h

↓ tl/pm

2 :: [1]

h [3 4 5] [2 1]

↓ tl/pm

3 :: [2 1]

h [4 5] [3 2 1]

↓ tl/pm

4 :: [3 2 1]

h [5] [4 3 2 1]

h [] [5 4 3 2 1]