Recursion

2024 Winter APS 105: Computer Fundamentals
Jon Eyolfson

Lecture 25
1.0.0

A Recursive Function Calls Itself

We need two things:

1. a base case: a simple solution we know
2. a recursive step: reduces the problem to a smaller version of itself

recursion:
  see *recursion*

Fibonacci Numbers Are An Example of Recursion

Consider the following recurrence relation:

$F_0 = 0$

$F_1 = 1$

and

$F_n = F_{n-1} + F_{n-2}$

for $n > 1$

Can we write a function to compute F_n?

Our Solution Calls Itself Twice in the Recursive Step

```
int fib(int n) {
    /* Base case */
    if (n < 2) {
        return n;
    }
    /* Recursive step */
    else {
        return fib(n - 1) + fib(n - 2);
    }
}
```

We'll re-visit this problem later

We Can Calculate Exponents Using Recursion As Well

Let's assume n can not be negative:

$$b^n = \underbrace{b \times b \times \ldots \times b \times b}_{\text{n times}}$$

What are the two things we need? What should they be?

We Can Calculate Exponents Using Recursion As Well

Let's assume n can not be negative:

$$b^n = \underbrace{b \times b \times \ldots \times b \times b}_{\text{n times}}$$

What are the two things we need? What should they be?

Base case: $b^0 = 1$

We Can Calculate Exponents Using Recursion As Well

Let's assume n can not be negative:

$$b^n = \underbrace{b \times b \times \dots \times b \times b}_{\text{n times}}$$

What are the two things we need? What should they be?

- Base case: $b^0 = 1$
- Recursive step: $b^n = b \times b^{n-1}$

Our Solution Calls Itself Once in the Recursive Step

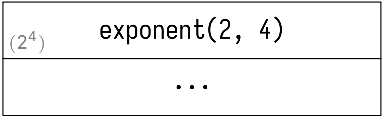
```
int exponent(int b, int n) {
    /* Base case */
    if (n == 0) {
        return 1;
    }
    /* Recursive step */
    else {
        return b * exponent(b, n - 1);
    }
}
```

Can you think of another way to calculate an exponent?

Let's Evaluate `exponent(2, 4)`

...

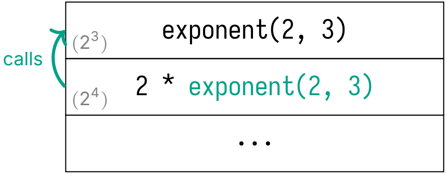
Let's Evaluate `exponent(2, 4)`



Let's Evaluate `exponent(2, 4)`

(2^4)	`2 * exponent(2, 3)`
	`...`

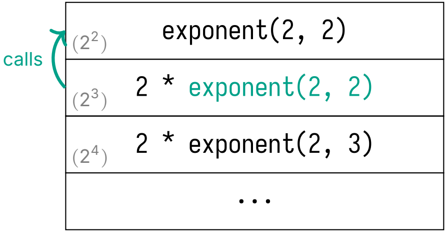
Let's Evaluate `exponent(2, 4)`



Let's Evaluate `exponent(2, 4)`

(2^3)	`2 * exponent(2, 2)`
(2^4)	`2 * exponent(2, 3)`
	...

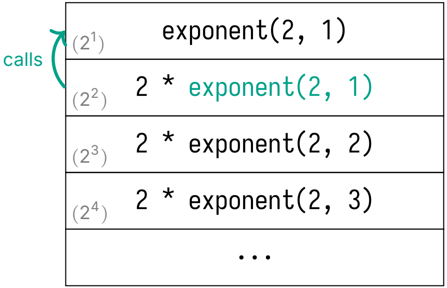
Let's Evaluate `exponent(2, 4)`



Let's Evaluate `exponent(2, 4)`

(2^2)	`2 * exponent(2, 1)`
(2^3)	`2 * exponent(2, 2)`
(2^4)	`2 * exponent(2, 3)`
	...

Let's Evaluate `exponent(2, 4)`



Let's Evaluate `exponent(2, 4)`

(2^1)	`2 * exponent(2, 0)`
(2^2)	`2 * exponent(2, 1)`
(2^3)	`2 * exponent(2, 2)`
(2^4)	`2 * exponent(2, 3)`
	`...`

Let's Evaluate `exponent(2, 4)`

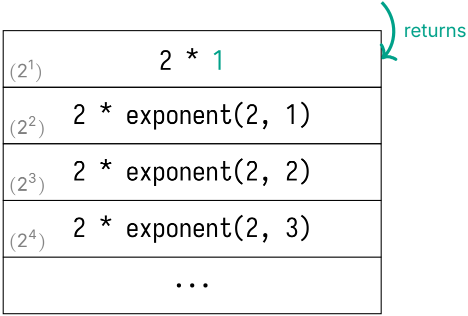
calls

(2^0)	exponent(2, 0)
(2^1)	2 * exponent(2, 0)
(2^2)	2 * exponent(2, 1)
(2^3)	2 * exponent(2, 2)
(2^4)	2 * exponent(2, 3)
	...

Let's Evaluate `exponent(2, 4)`

(2^0)	`1`
(2^1)	`2 * exponent(2, 0)`
(2^2)	`2 * exponent(2, 1)`
(2^3)	`2 * exponent(2, 2)`
(2^4)	`2 * exponent(2, 3)`
	`...`

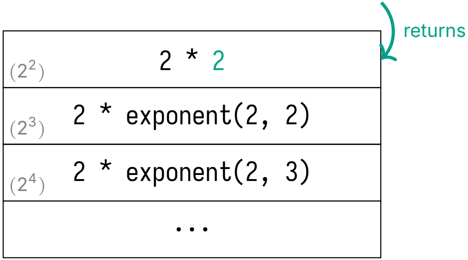
Let's Evaluate `exponent(2, 4)`



Let's Evaluate `exponent(2, 4)`

(2^1)	2
(2^2)	`2 * exponent(2, 1)`
(2^3)	`2 * exponent(2, 2)`
(2^4)	`2 * exponent(2, 3)`
	...

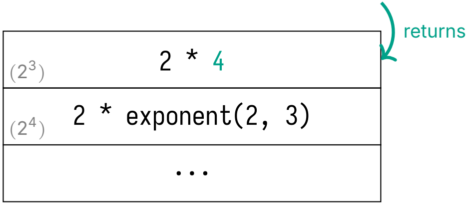
Let's Evaluate `exponent(2, 4)`



Let's Evaluate `exponent(2, 4)`

(2^2)	`4`
(2^3)	`2 * exponent(2, 2)`
(2^4)	`2 * exponent(2, 3)`
	`...`

Let's Evaluate `exponent(2, 4)`



Let's Evaluate `exponent(2, 4)`

(2^3)	8
(2^4)	`2 * exponent(2, 3)`
	...

Let's Evaluate `exponent(2, 4)`

Let's Evaluate `exponent(2, 4)`

(2^4) 16
. . .

We Can Run Out of Memory and Cause a "Stack Overflow"

Every time we call a function, that function has its own copy of variables
- C stores local variables on the stack

There may be many functions active at once, we could run out of memory
- Running out of memory for local variables is a stack overflow

We Can Run Out of Memory and Cause a "Stack Overflow"

Every time we call a function, that function has its own copy of variables
- C stores local variables on the stack

There may be many functions active at once, we could run out of memory
- Running out of memory for local variables is a stack overflow

A common cause for a stack overflow is infinite recursion
- Similar to an infinite loop

We Can Re-write Some Recursive Functions to Use “Tail Recursion”

A tail recursive function has a single recursive call in the return statement

- This is beyond the scope of this course

We can modify `exponent` to follow this form by creating another function:

```
int exponent_tail(int accumulator, int b, int n) {
    /* Base case */
    if (n == 0) { return accumulator; }
    /* Recursive case */
    else        { return exponent_tail(b * accumulator, b, n - 1); }
}

int exponent(int b, int n) {
    return exponent_tail(1, b, n);
}
```

Compilers Can Optimize Tail Recursive Functions

If we turn on compiler optimizations, it'll convert `exponent_tail` to:

```
int exponent_tail(int accumulator, int b, int n) {
    int x = accumulator;
    while (n != 0) {
        x *= b;
        n = n - 1;
    }
    return x;
}
```

Now our code will not have a stack overflow

- We're still able to keep the (maybe) more readable recursive solution

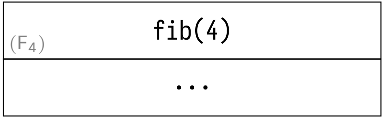
What Happens When We Call `fib(4)`?

```c
int fib(int n) {
    /* Base case */
    if (n < 2) {
        return n;
    }
    /* Recursive step */
    else {
        return fib(n - 1) + fib(n - 2);
    }
}
```

Let's Evaluate `fib(4)`

...

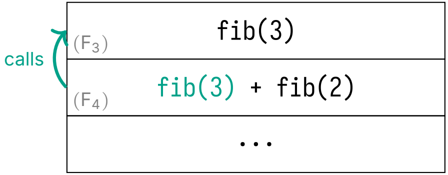
Let's Evaluate `fib(4)`



Let's Evaluate `fib(4)`

(F_4)	`fib(3) + fib(2)`
	...

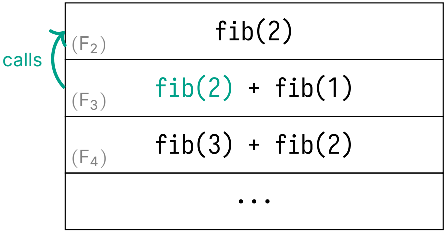
Let's Evaluate `fib(4)`



Let's Evaluate `fib(4)`

(F_3)	`fib(2) + fib(1)`
(F_4)	`fib(3) + fib(2)`
	...

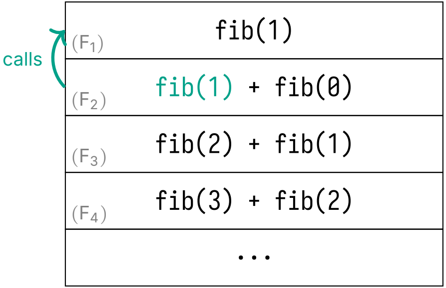
Let's Evaluate `fib(4)`



Let's Evaluate `fib(4)`

(F_2)	`fib(1) + fib(0)`
(F_3)	`fib(2) + fib(1)`
(F_4)	`fib(3) + fib(2)`
	...

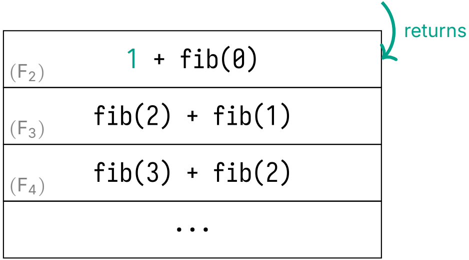
Let's Evaluate `fib(4)`



Let's Evaluate `fib(4)`

(F_1)	`1`
(F_2)	`fib(1) + fib(0)`
(F_3)	`fib(2) + fib(1)`
(F_4)	`fib(3) + fib(2)`
	`...`

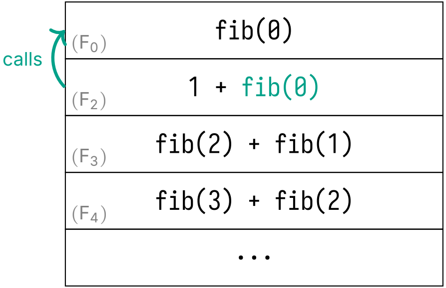
Let's Evaluate `fib(4)`



Let's Evaluate `fib(4)`

(F_2)	`1 + fib(0)`
(F_3)	`fib(2) + fib(1)`
(F_4)	`fib(3) + fib(2)`
	...

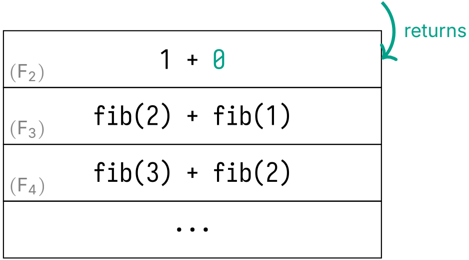
Let's Evaluate `fib(4)`



Let's Evaluate `fib(4)`

(F_0)	`0`
(F_2)	`1 + fib(0)`
(F_3)	`fib(2) + fib(1)`
(F_4)	`fib(3) + fib(2)`
	...

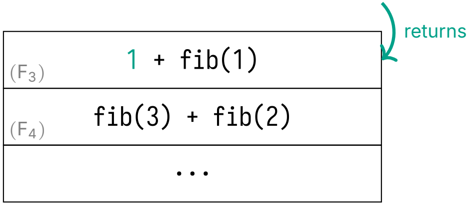
Let's Evaluate `fib(4)`



Let's Evaluate `fib(4)`

(F_2)	1
(F_3)	`fib(2) + fib(1)`
(F_4)	`fib(3) + fib(2)`
	...

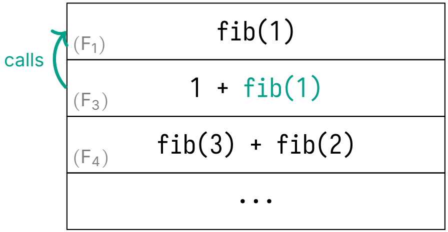
Let's Evaluate `fib(4)`



Let's Evaluate `fib(4)`

(F_3)	`1 + fib(1)`
(F_4)	`fib(3) + fib(2)`
	...

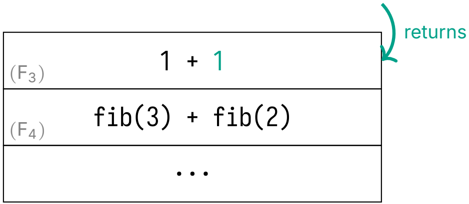
Let's Evaluate `fib(4)`



Let's Evaluate `fib(4)`

(F_1)	`1`
(F_3)	`1 + fib(1)`
(F_4)	`fib(3) + fib(2)`
	`...`

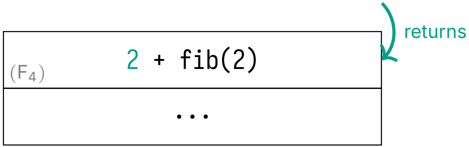
Let's Evaluate `fib(4)`



Let's Evaluate `fib(4)`

(F_3)	2
(F_4)	fib(3) + fib(2)
	...

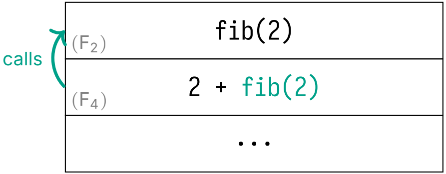
Let's Evaluate `fib(4)`



Let's Evaluate `fib(4)`

(F_4)	`2 + fib(2)`
	...

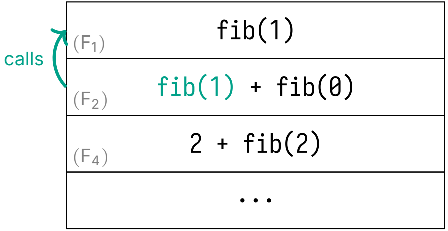
Let's Evaluate `fib(4)`



Let's Evaluate `fib(4)`

(F_2)	`fib(1) + fib(0)`
(F_4)	`2 + fib(2)`
	...

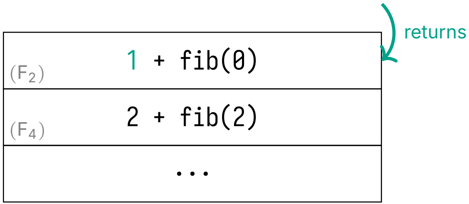
Let's Evaluate `fib(4)`



Let's Evaluate `fib(4)`

(F_1)	`1`
(F_2)	`fib(1) + fib(0)`
(F_4)	`2 + fib(2)`
	`...`

Let's Evaluate `fib(4)`



Let's Evaluate `fib(4)`

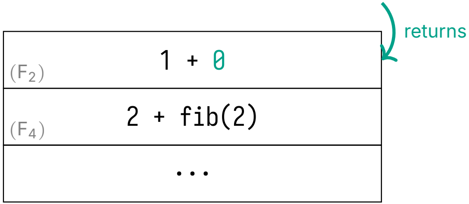
(F_2)	`1 + fib(0)`
(F_4)	`2 + fib(2)`
	...

Let's Evaluate `fib(4)`

Let's Evaluate `fib(4)`

(F_0)	`0`
(F_2)	`1 + fib(0)`
(F_4)	`2 + fib(2)`
	`...`

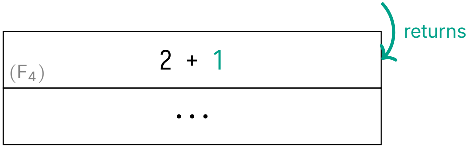
Let's Evaluate `fib(4)`



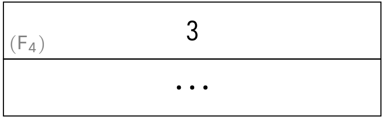
Let's Evaluate `fib(4)`

(F_2)	1
(F_4)	`2 + fib(2)`
	...

Let’s Evaluate `fib(4)`



Let's Evaluate `fib(4)`



We Can Significantly Speed Up `fib` With Memoization

Memoization is just a fancy word for caching

- Caching is saving values so you don't have to re-compute them
- Again, this part is beyond the scope of this course

`fib` includes a lot of repeated calls we already computed

Optimization: remember the value, and re-use it instead of re-computing

Recursive Functions are Just Another Tool

Some problems are easier to solve recursively

Typically, recursive functions require more space to execute
- Tail recursive functions can be optimized

It’s important to practice, so you can identify these problems

Can We Write a Recursive Function to Calculate Factorials?

Write a function, `int factorial(int n)`, to compute n!

e.g. $4! = 4 \times 3 \times 2 \times 1$

Example Recursive Function to Compute Factorials

```c
int factorial(int n) {
    /* Base case */
    if (n < 2) {
        return 1;
    }
    /* Recursive step */
    else {
        return n * factorial(n - 1);
    }
}
```

A Recursive Function Calls Itself

We need two things:

1. a base case: a simple solution we know
2. a recursive step: reduces the problem to a smaller version of itself