Programming Abstractions

CS106B

Instructor: Cynthia Bailey
Head TA: Yasmine Alonso

pollev.com/cs106b

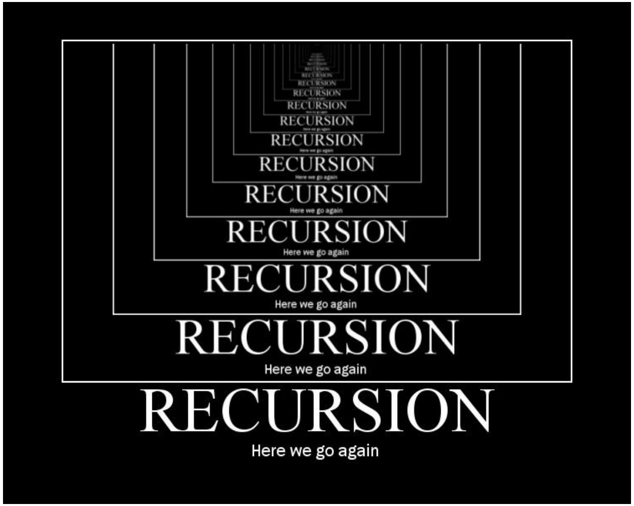
Today's Topics

- Recursion!
  - › Functions calling functions
- Next time:
  - › More recursion! It's Recursion Week!
  - › Like Shark Week, but more nerdy

- **For important announcements, be sure to see the weekly announcements post on the Ed Q&A board! https://edstem.org**
- **Also on Ed: live lecture Q&A**

Quick Course Overview

- Week 1: C++
- Week 2: ADTs, *how to use them*
- Weeks 3-4: Recursion □ **YOU ARE HERE**
- Weeks 5-10: ADTs, *behind the scenes!*
  - actually implement them yourself



Recursion!

The exclamation point isn't there only because this is so exciting; it also relates to our first recursion example….

Factorial!

$$n! = n(n-1)(n-2)(n-3)(n-4)\ldots(3)(2)(1)$$

This could be a really long expression!

Recursion is a technique for tackling large or complicated problems by just "eating" one "bite" of the problem at a time.

Factorial!

$$n! = n(n-1)(n-2)(n-3)(n-4)\ldots(2)(1)$$

An alternate mathematical formulation:

$$n! = \begin{cases} 1 & \textit{if } n = 1 \\ n(n-1)! & \textit{otherwise} \end{cases}$$

Translated to code

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

Factorial!

$$n! = n(n-1)(n-2)(n-3)(n-4)\ldots(2)(1)$$

An alternate mathematical formulation:

$$n! = \begin{cases} 1 & \text{if } n = 1 \\ n(n-1)! & \text{otherwise} \end{cases}$$

Debugging tip:
When reading your own code, mentally assume the recursive call works perfectly, and reason from there.

Translated to code

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial_teacher_solution(n - 1);
    }
}
```

Basic Recursive Function Design Pattern

Always two parts:

Base case:

- This problem is so tiny, it's hardly a problem anymore! Just give answer.

Recursive case:

- This problem is still a bit large, let's (1) bite off just one piece, and (2) delegate the remaining work to recursion.

The recursive function pattern

Recursive case:

- This problem is still a bit large, let's **(1) bite off just one piece**, and (2) delegate the remaining work to recursion.

```
int factorial(int n) {
    if (n == 1) {  // Easy! Return trivial answer
        return 1;
    } else {  // Not easy enough to finish yet!
        return n * factorial(n - 1);
    }
}
```

Do <u>one</u> of the many, many multiplications required for factorial.

The recursive function pattern

Recursive case:

- This problem is still a bit large, let's (1) bite off just one piece, and **(2) delegate the remaining work to recursion**.

```
int factorial(int n) {
    if (n == 1) {  // Easy! Return trivial answer
        return 1;
    } else {  // Not easy enough to finish yet!
        return n * factorial(n - 1);
    }
}
```

Do one of the many, many multiplications required for factorial.

Delegate all the other multiplications to the recursive call.

Digging deeper in the recursion

Looking at how recursion works “under the hood”

Factorial!

```
int factorial(int n) {
    cout << n << endl;  // **Added for this question**
    if (n == 1) {  // Easy! Return trivial answer
        return 1;
    } else {  // Not easy enough to finish yet!
        return n * factorial(n - 1);
    }
}
```

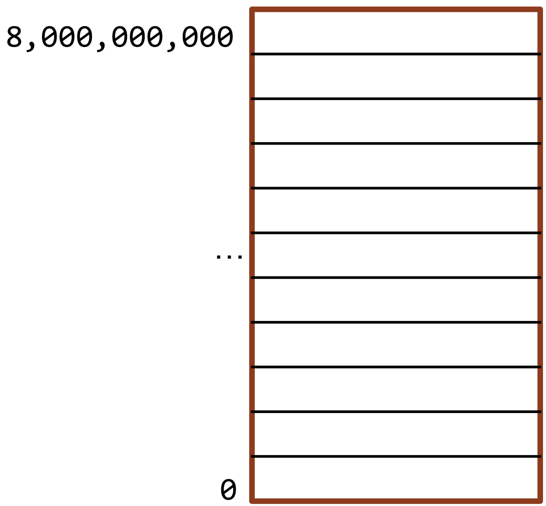
What is the **third** thing **printed** when we call `factorial(4)`?

A. 1
B. 2
C. 3
D. 4
E. Other/none/more

pollev.com/cs106b

How does this look in memory? A little background…

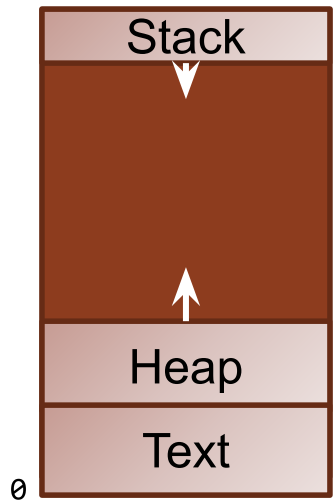
- A computer's memory is like a giant Vector/array, and like a Vector, we start counting at index `0`.
- We typically draw memory vertically (rather than horizontally like a Vector), with index `0` at the bottom.
- A typical laptop's memory has billions of these indexed slots (one byte each)



* Take CS107 to learn much more!!

How does this look in memory?
A little background…

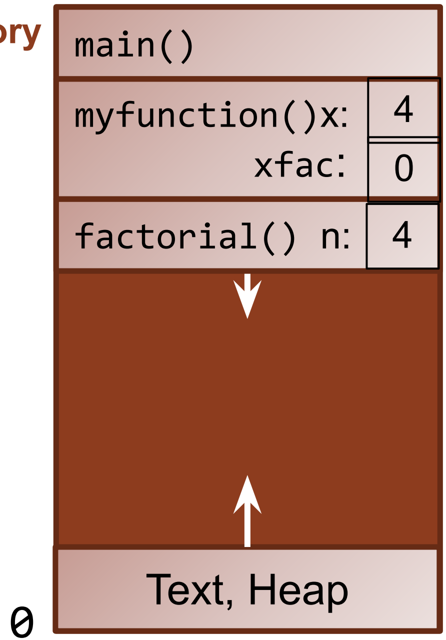
- Broadly speaking, we divide memory into regions:
  - **Text:** the program's own code (needs to be in memory so it can run!)
  - **Heap:** we'll learn about this later in CS106B!
  - **Stack:** this is where local variables for each function are stored.



* Take CS107 to learn much more!!

How does this look in memory?

Memory

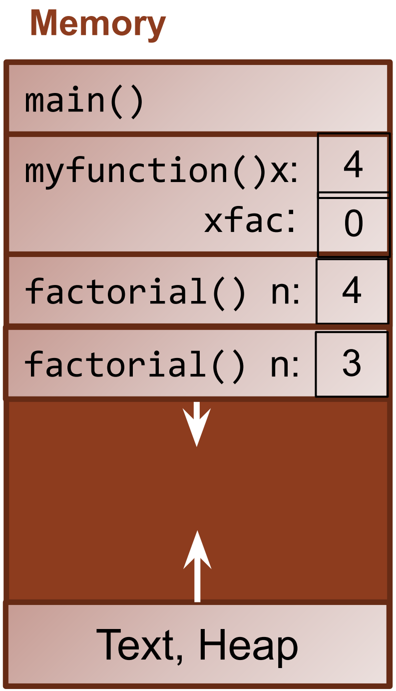


Recursive code

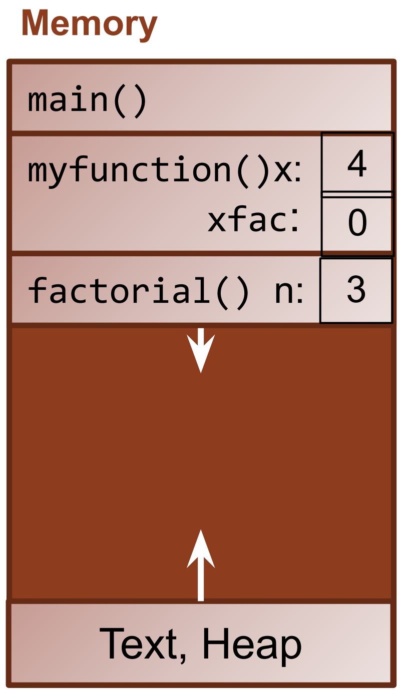
```
int factorial(int n) {
    cout << n << endl;
    if (n == 1) {
            return 1;
    } else {
            return n * factorial(n – 1);
     }
}

void myfunction(){
     int x = 4;
     int xfac = 0;
     xfac = factorial(x);
}
```

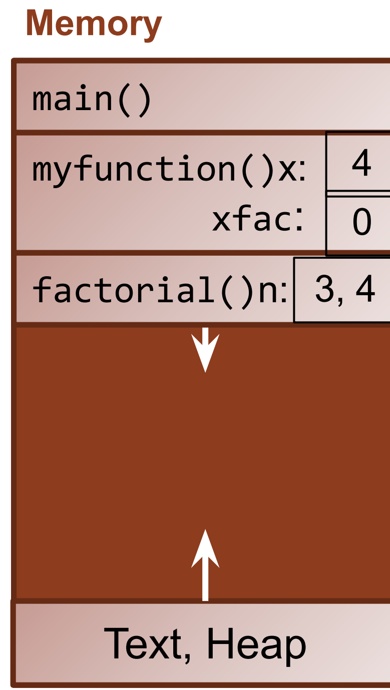
pollev.com/cs106b



(A)



(B)



(C)

(D) Other/none of the above

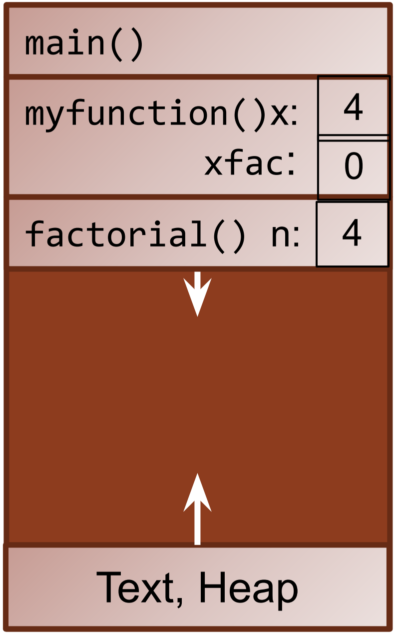
Fun fact:
The “stack” part of memory is a stack

Function **call** = **push** a stack frame

Function **return** = **pop** a stack frame

* Take CS107 to learn much more!!

The “stack” part of memory is a stack

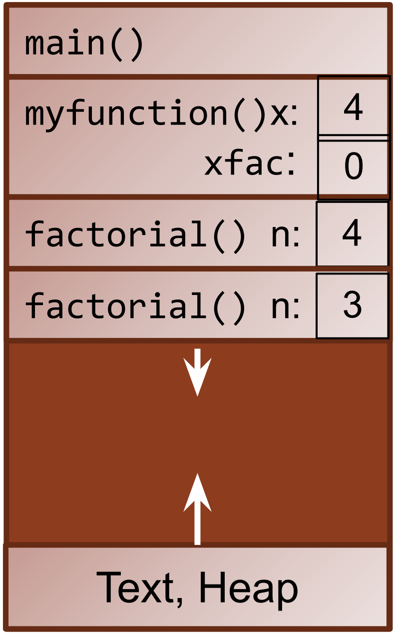


Recursive code

```
int factorial(int n) {
    cout << n << endl;
    if (n == 1) {
            return 1;
    } else {
            return n * factorial(n – 1);
     }
}

void myfunction(){
      int x = 4;
      int xfac = 0;
      xfac = factorial(x);
}
```

The “stack” part of memory is a stack

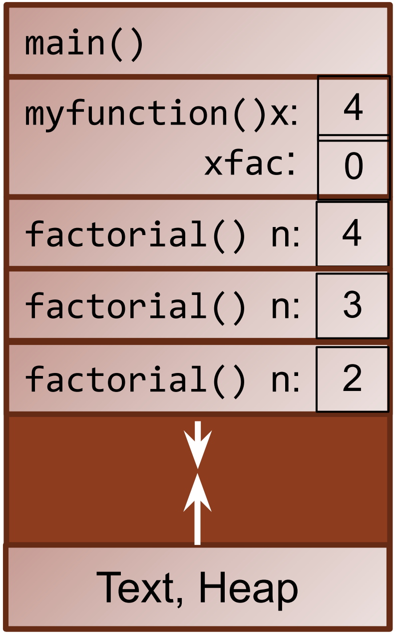


Recursive code

```
int factorial(int n) {
    cout << n << endl;
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

void myfunction(){
    int x = 4;
    int xfac = 0;
    xfac = factorial(x);
}
```

The “stack” part of memory is a stack



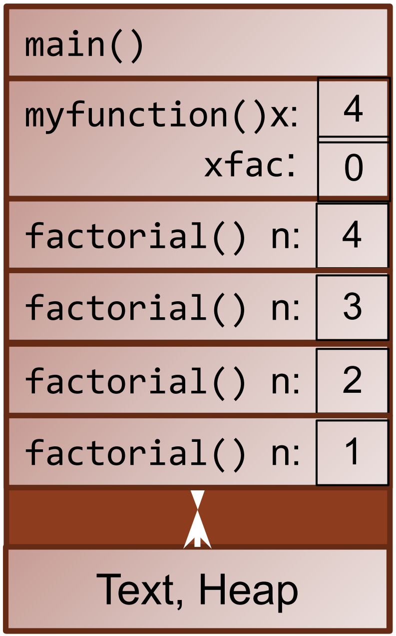
Recursive code

Answer: 3rd thing printed is 2

```
int factorial(int n) {
    cout << n << endl;
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

void myfunction(){
    int x = 4;
    int xfac = 0;
    xfac = factorial(x);
}
```

The “stack” part of memory is a stack



Recursive code

```
int factorial(int n) {
    cout << n << endl;
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

void myfunction(){
    int x = 4;
    int xfac = 0;
    xfac = factorial(x);
}
```

Factorial!

What is the **fourth** value ever **returned** when we call `factorial(4)`?

A. 4
B. 6
C. 10
D. 24
E. Other/none/more than one

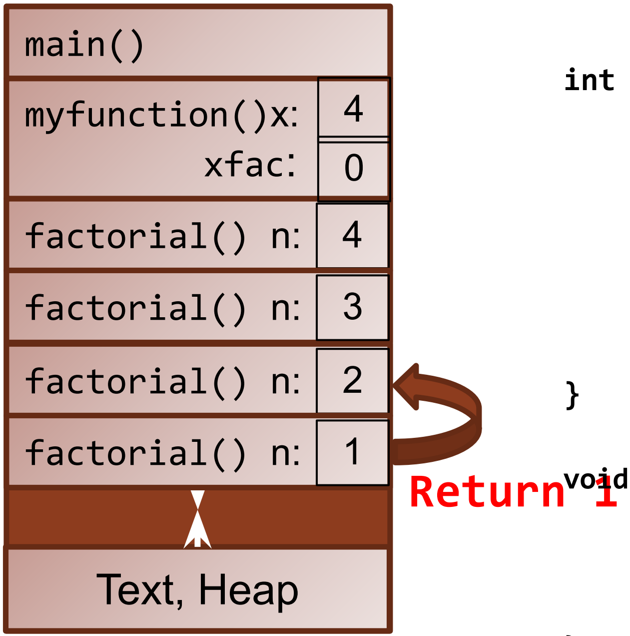
Recursive code

```
int factorial(int n) {
    cout << n << endl;
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

void myfunction(){
    int x = 4;
    int xfac = 0;
    xfac = factorial(x);
}
```

pollev.com/cs106b

The “stack” part of memory is a stack

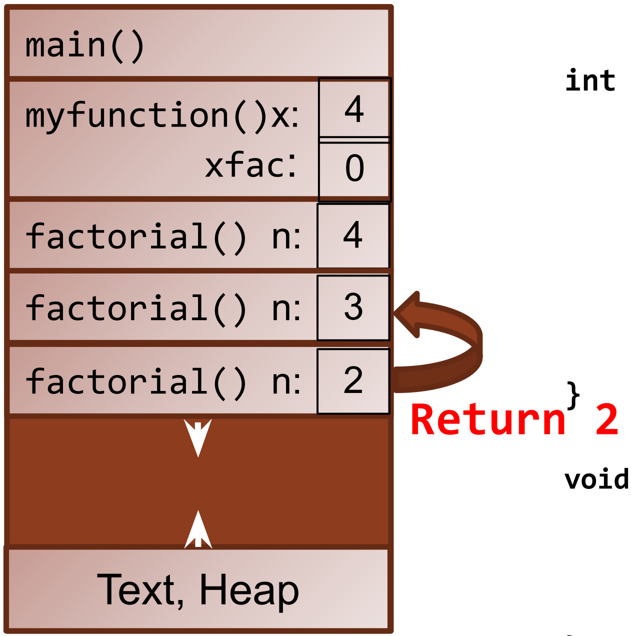


Recursive code

```
int factorial(int n) {
    cout << n << endl;
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

void myfunction(){
    int x = 4;
    int xfac = 0;
    xfac = factorial(x);
}
```

The “stack” part of memory is a stack

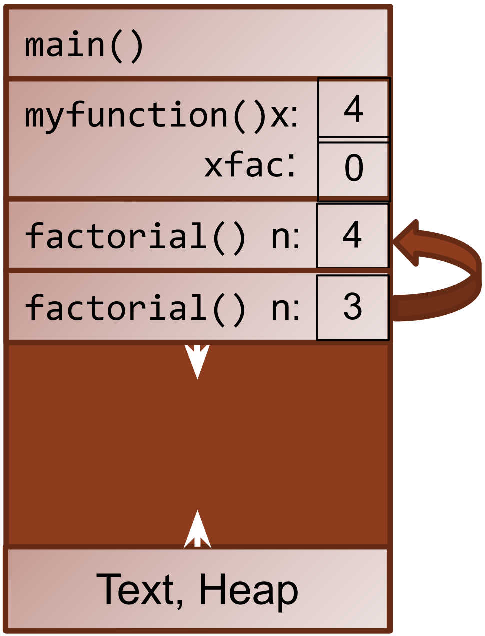


Recursive code

```
int factorial(int n) {
    cout << n << endl;
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

void myfunction(){
    int x = 4;
    int xfac = 0;
    xfac = factorial(x);
}
```

The “stack” part of memory is a stack



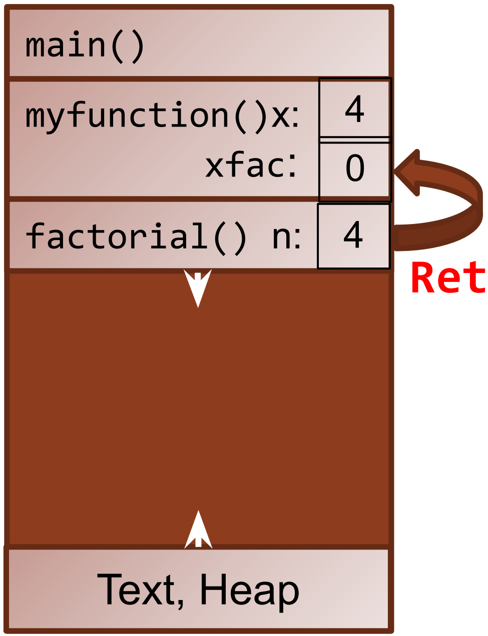
Return 6

Recursive code

```
int factorial(int n) {
    cout << n << endl;
    if (n == 1) {
            return 1;
    } else {
            return n * factorial(n – 1);
     }
}

void myfunction(){
      int x = 4;
      int xfac = 0;
      xfac = factorial(x);
}
```

The “stack” part of memory is a stack



Return 24

Recursive code

```
int factorial(int n) {
    cout << n << endl;
    if (n == 1) {
        return 1;
    } else {
        return n                1);
    }
}

void myfunction(){
    int x = 4;
    int xfac = 0;
    xfac = factorial(x);
}
```

Answer: 4th thing returned is 24

Factorial!

Iterative version

```
int factorial(int n) {
    int f = 1;
    while (n > 1) {
        f = f * n;
        n = n - 1;
    }
    return f;
}
```

Recursive version

```
int factorial(int n) {
    cout << n << endl;
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Factorial!

Iterative version

```
int factorial(int n) {
    int f = 1;
    while (n > 1) {
        f = f * n;
        n = n - 1;
    }
    ret
}
```

Recursive version

```
int factorial(int n) {
    cout << n << endl;
    if (n == 1) {
        return 1;
    } else {
                        (n - 1);
```

Performance efficiency remarks:

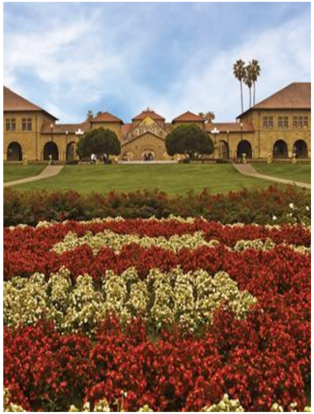
Iterative is sometimes much faster because it doesn't have to push and pop stack frames. Function calls have overhead in terms of space *and* time (to set up and tear down).

But recursive code is often much more elegant (not really in this case, but we'll see others).

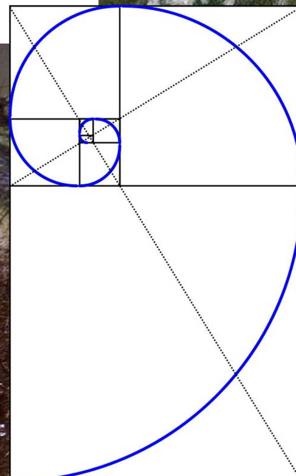
Choose elegance, and the C++ compiler will save you on performance behind the scenes, by secretly rewriting your recursion as iterative code, in cases where it would help.

The Fibonacci Sequence

MATH NERD REJOICING INTENSIFIES

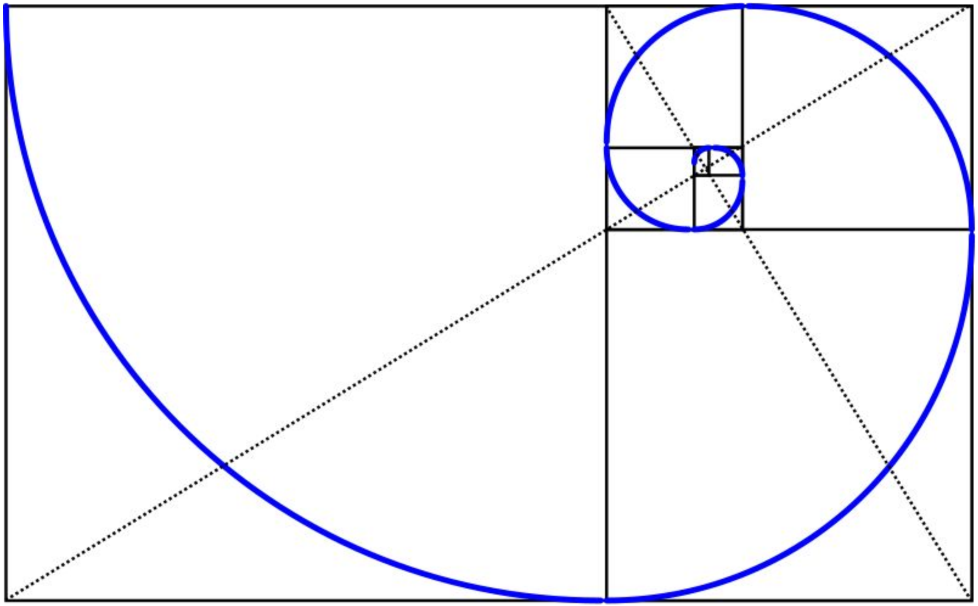


Fibonacci in nature



Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,



Fibonacci

0	1	2	3	4	5	6	7	8	9	10	11
0	1	1	2	3	5	8	13	21	34	55	89

$$Fib_N = \begin{cases} 0 & if\ N = 0 \\ 1 & if\ N = 1 \\ Fib_{N-1} + Fib_{N-2} & otherwise \end{cases}$$

Basic Recursive Function Design Pattern

Always two parts:

Base case:

- This problem is so tiny, it's hardly a problem anymore! Just give answer.

Recursive case:

- This problem is still a bit large, let's (1) bite off just one piece, and (2) delegate the remaining work to recursion.

```
int fibonacci(int n)
{
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fibonacci(n – 1) + fibonacci(n – 2);
    }
}
```

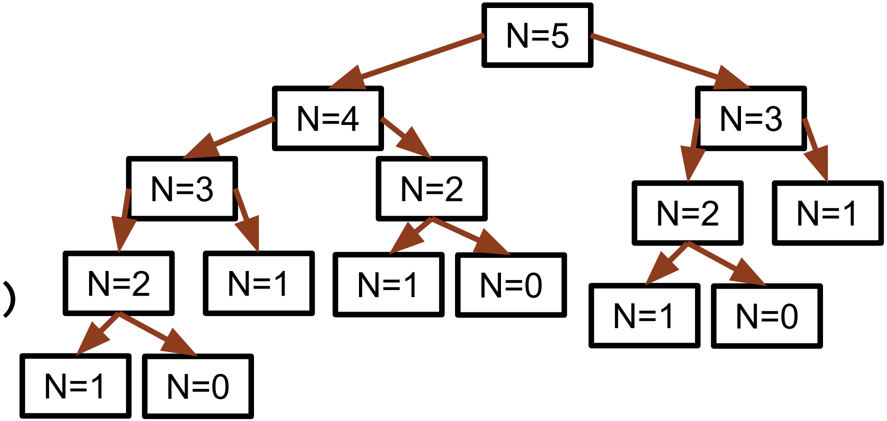
$$Fib_N = \begin{cases} 0 & if\ N = 0 \\ 1 & if\ N = 1 \\ Fib_{N-1} + Fib_{N-2} & otherwise \end{cases}$$

Fibonacci

```
int fibonacci(int n)
{
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fibonacci(n – 1) + fibonacci(n – 2);
    }
}
```

- Now that we have code that works, let's consider this code's efficiency using Big-O analysis.
- We will start by doing a precise-ish count of **how many function calls** happen.

Fibonacci



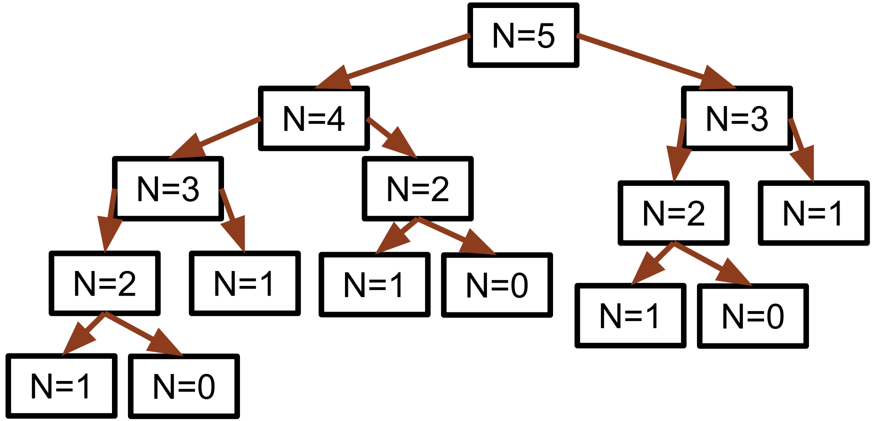

```
int fibonacci(int n)
{
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

Observation: lots of redundant work happens throughout the call tree

- fibonacci(2) is calculated 3 separate times when calculating fibonacci(5)!
- 15 function calls in total for fibonacci(5)!

Fibonacci

`fibonacci(2)` is calculated 3 separate times when calculating `fibonacci(5)`!



How many times would we calculate `fibonacci(2)` while calculating `fibonacci(6)`?

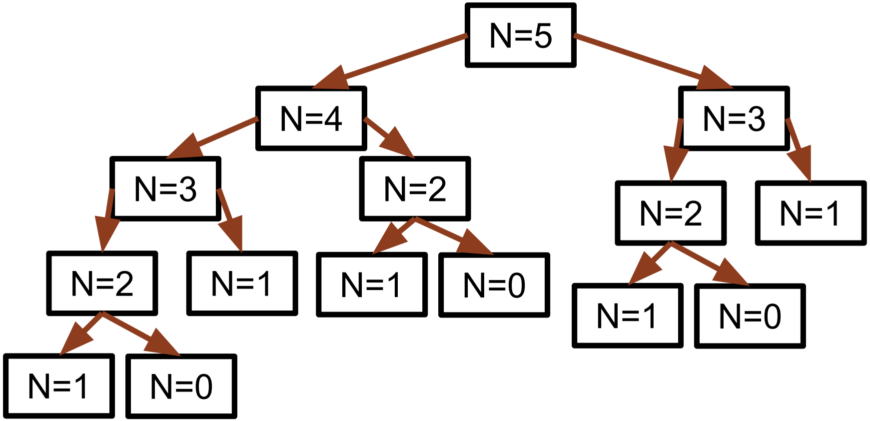
See if you can just "read" it off the chart above.

A. 4 times
B. 5 times
C. 6 times
D. Other/none/more

pollev.com/cs106b

Fibonacci

N	fib(N)	# of calls to fib(2)
2	1	1
3	2	1
4	3	2
5	5	3
6	8	5
7	13	8
8	21	13
9	34	21
10	55	34



Big-O of naïve Fibonacci implementation

When we **added 1** to the input N, the number of times we had to calculate fibonacci(2) **nearly doubled** (~1.6* times)

- Ouch!

Goal: predict how much time it will take to compute for arbitrary input N.

Calculation (approximate):

N times

- $\underbrace{(1.6)(1.6)(1.6)\ldots(1.6)(1.6)(1.6)}_{N \text{ times}}$
- = $\mathbf{O(1.6)^N}$

* This ~1.6 number is called the "Golden Ratio" in math—cool!