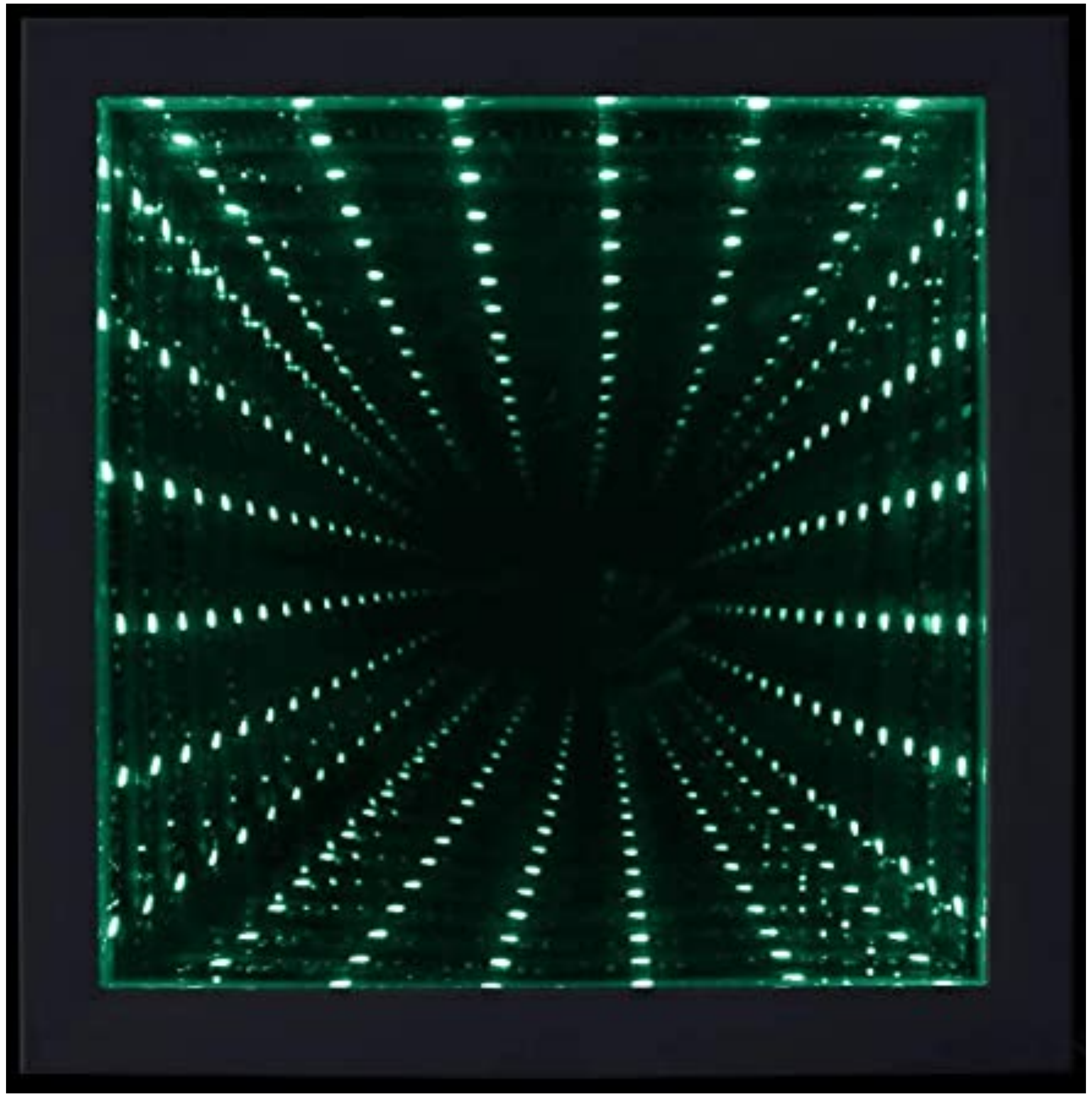
Fun with Recursion

with Fun with Recursion

with Fun with Recursion…



Docrob

What is recursion?

- A function that calls itself
- But this definition is pretty narrow
  - focuses on just AN implementation of a solution for a certain kind of problem

What is a recursive problem?

OR: what kinds of problems can recursion solve

- A problem whose definition (or answer or solution) is expressed in <u>**smaller**</u> **terms of itself**

Requirements for a recursive solution

(or definition of a solution)

1. Answer is expressed in SMALLER terms of itself
2. Answer has an escape clause, i.e., a way to STOP calling itself

These are also called the HALLMARKS of a recursive problem

Pros of recursion

- In general: WAY MORE READABLE
- A recursive definition is a near perfect roadmap for implementing the solution, i.e., programming it
  - The code looks JUST LIKE the recursive definition
- Takes less code compared to other solutions

Cons of recursion

- Tend to be computationally inefficient (more on this later)
- Can kill your program (more on this later)
- Coming up with a recursive definition of a solution can be REALLY HARD

Example: factorial

- Problem description (from Wikipedia):

  n factorial (or ***n!***) is the product of all positive integers less than or equal to ***n***

Example: factorial

Coming up with the recursive definition

- Analysis of an example:

  5! = 5 x 4 x 3 x 2 x 1

- What do you notice about the above description of 5! ?
- It can be rewritten as 5! = 5 x 4!, or 5 x 4 x 3!, …
- Thus: n! = n x (n - 1)!

Answer is expressed in smaller terms of itself

Is there a way for it to STOP calling itself?

Yes, when n = 1 or 2

BINGO: we have a recursive definition!

Example: factorial

The recursive definition

- Hallmark 1: $n! = n \times (n - 1)!$
- Hallmark 2: $1! = 1$, $2! = 2$
  - Note: it is ok to just have $1! = 1$ for the escape clause

Example: factorial

Implementation of the recursive definition

Once we have the recursive definition, the code is easy:

```
public static long fact(long n) {
    // hallmark #2 (a way to stop)
    if(n == 1 || n == 2) {
        return n;
    }

    // hallmark #1 (answer is in smaller terms of itself)
    return n * fact(n - 1);
}
```

Is this readable?
Does it look like the recursive definition?

But… no tool is perfect

Enter fibonacci

- The recursive definition of fibonacci:
  - fib(n) = fib(n - 1) + fib(n - 2)
  - fib(0) = 0 and fib(1) = 1

Hallmark #1

Hallmark #2

Example: fibonacci

The naive implementation

```
public static long fib(int n) {
    // hallmark #2
    if(n == 0) {
        return 0;
    }
    if(n == 1) {
        return 1;
    }

    // hallmark #1
    return fib(n - 1) + fib(n - 2);
}
```

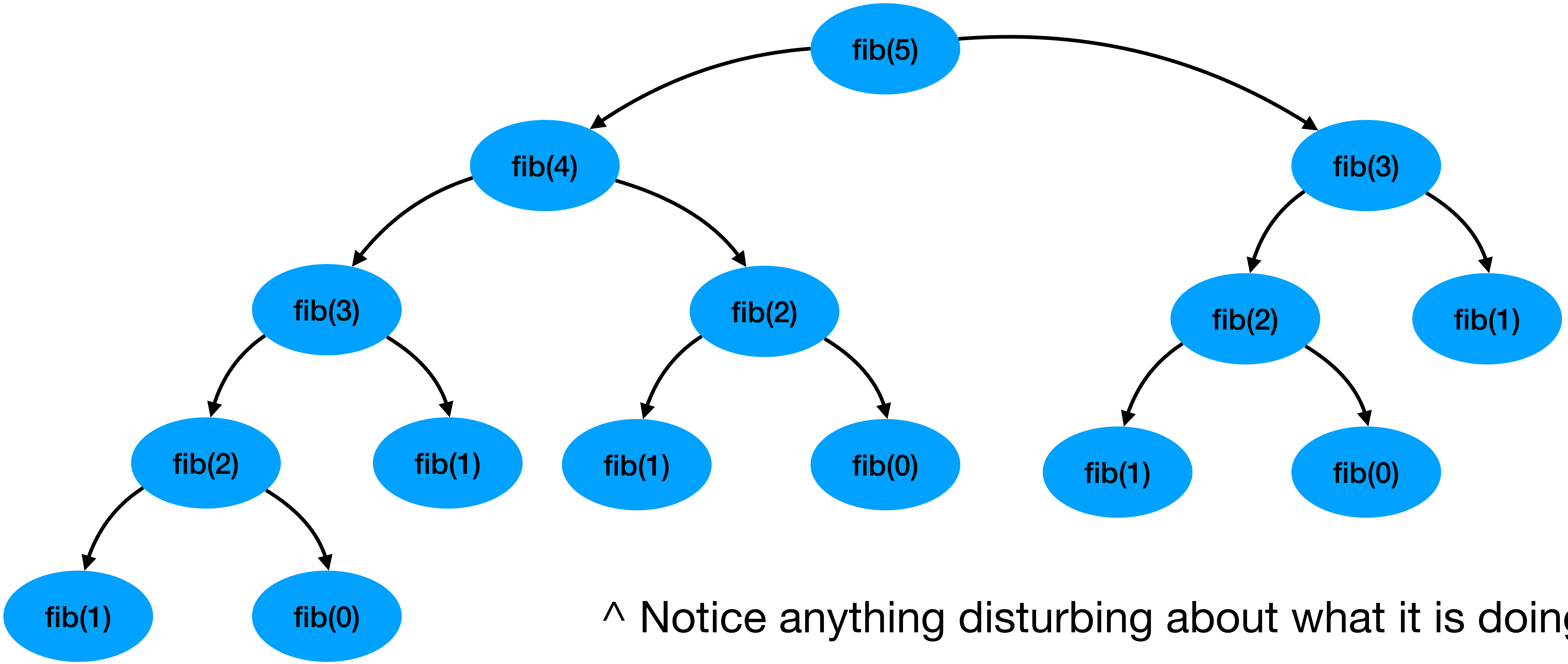
Try fib(10), fib(20), fib(30), fib(40), fib(50)

What do you notice???

Reminder: `return fib(n - 1) + fib(n - 2);`

Example: naive fibonacci

Let's analyze its performance! How many times does fib(5) call itself???

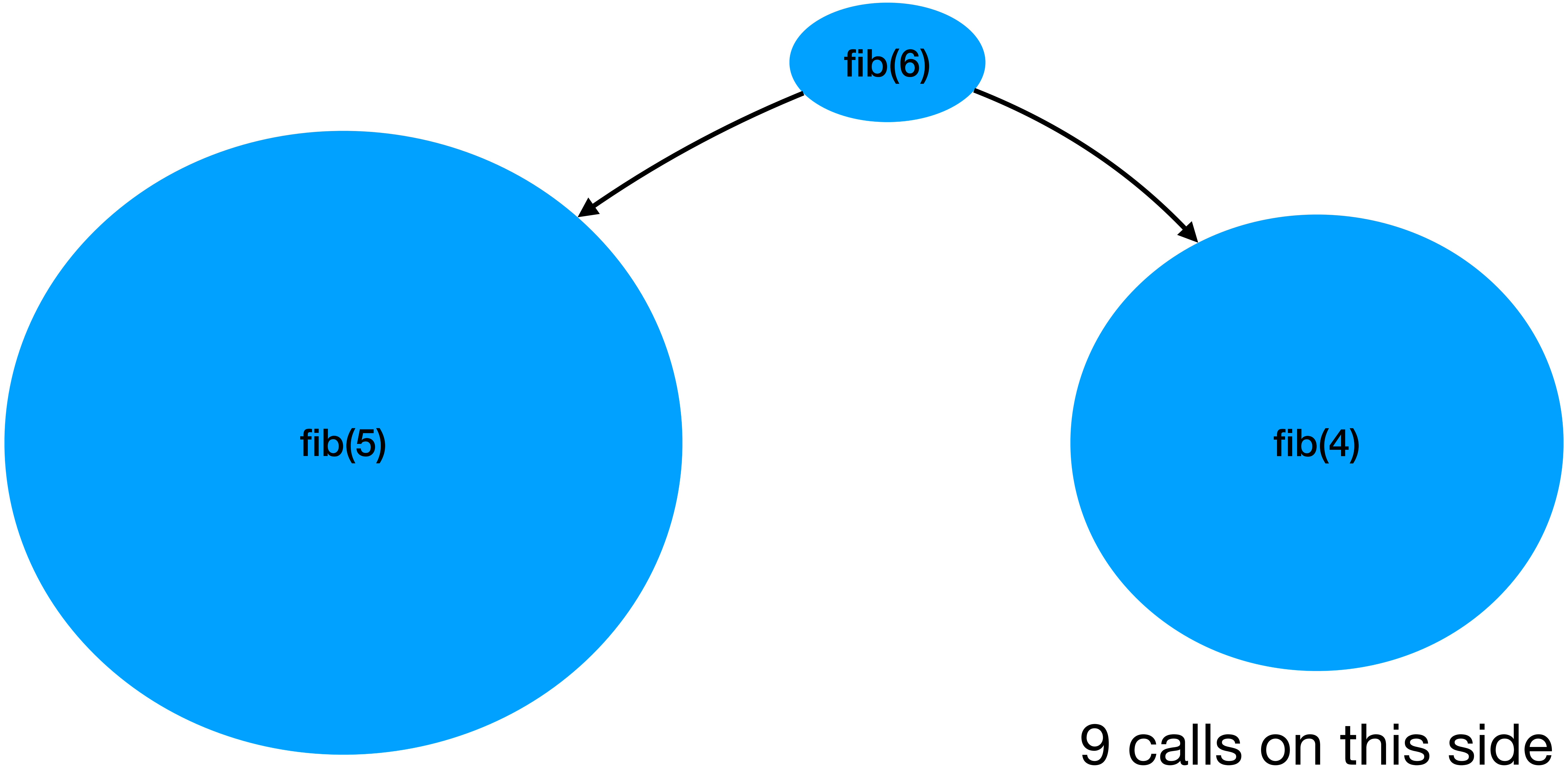


^ Notice anything disturbing about what it is doing???

Reminder: `return fib(n - 1) + fib(n - 2);`

Example: naive fibonacci

Let's analyze its performance! How many times does fib(6) call itself???



9 calls on this side

15 calls on this side

Example: naive fibonacci

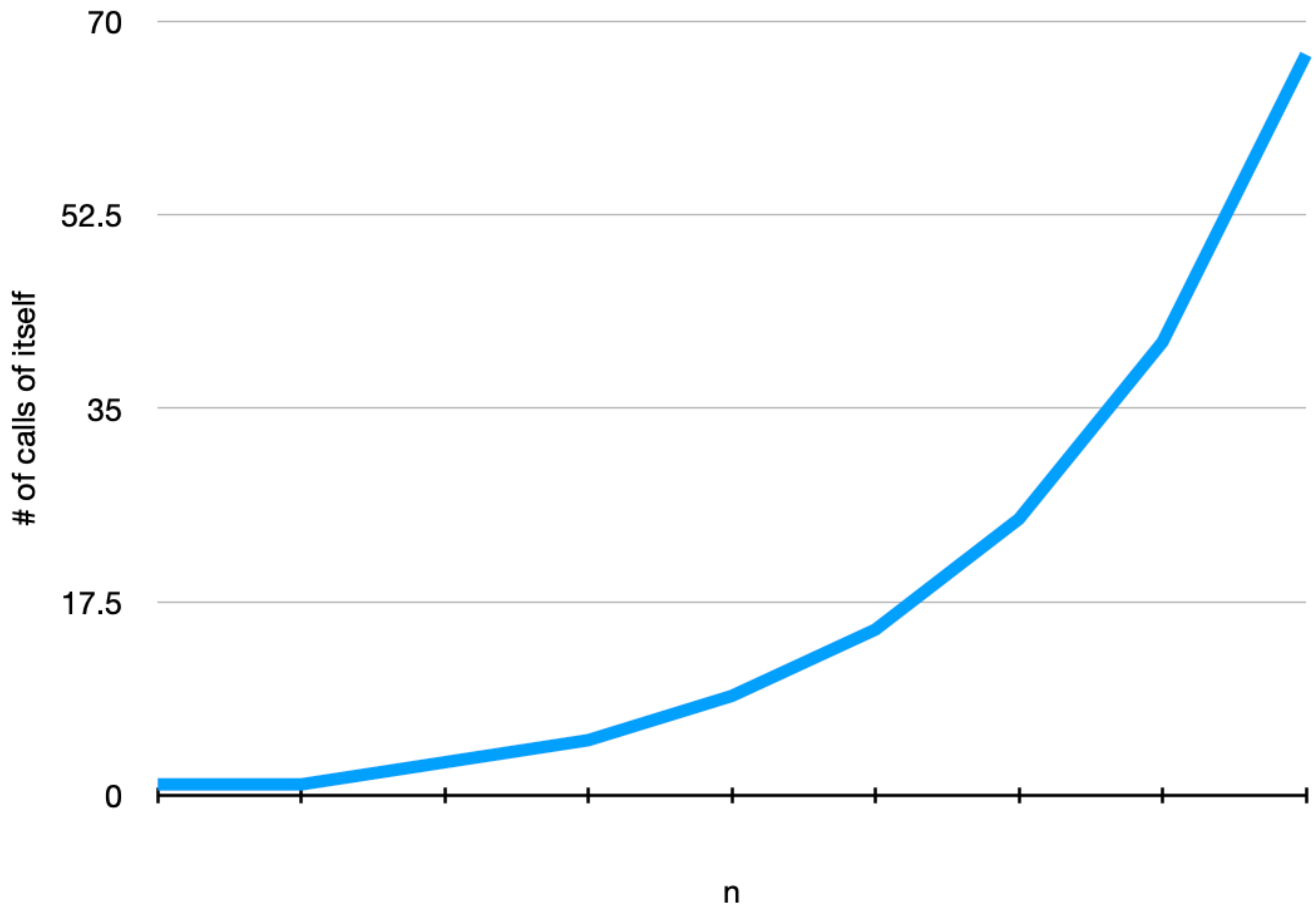
Performance analysis

- fib(4): 9 calls
- fib(5): 15 calls
- fib(6): 25 calls
- fib(7): 41 calls
- fib(8): 67 calls

… Time for a graph!

Example: naive fibonacci

Performance analysis



Every change in n results in a LOT more calls to fib().

This kind of computational growth in an algorithm is UNDESIRABLE.

As a result, we look for different algorithms and/or improvements to this one.

Example: fibonacci + memoization

Naive plus reuse previous calculations

- Don't recalculate things that have already been calculated
- When fib(n) is called, lookup n in some kind of array to see if it has already been calculated
  - **If yes**, return that value
  - **If no**, calculate it normally and then save it before returning it

Example: fibonacci + memoization

```
private static long [] memoTable = new long[MAX_FIB_NUMBER];

public static long fib(int n) {
    if(n == 0) {
        return 0;
    }
    if(n == 1) {
        return 1;
    }

    // if we have already calculated fib(n) then just return it
    if(memoTable[n] != 0) {
        return memoTable[n];
    }

    long fibN = fib(n - 1) + fib(n - 2);

    // save fib n to the memoTable for later reuse
    memoTable[n] = fibN;

    return fibN;
}
```

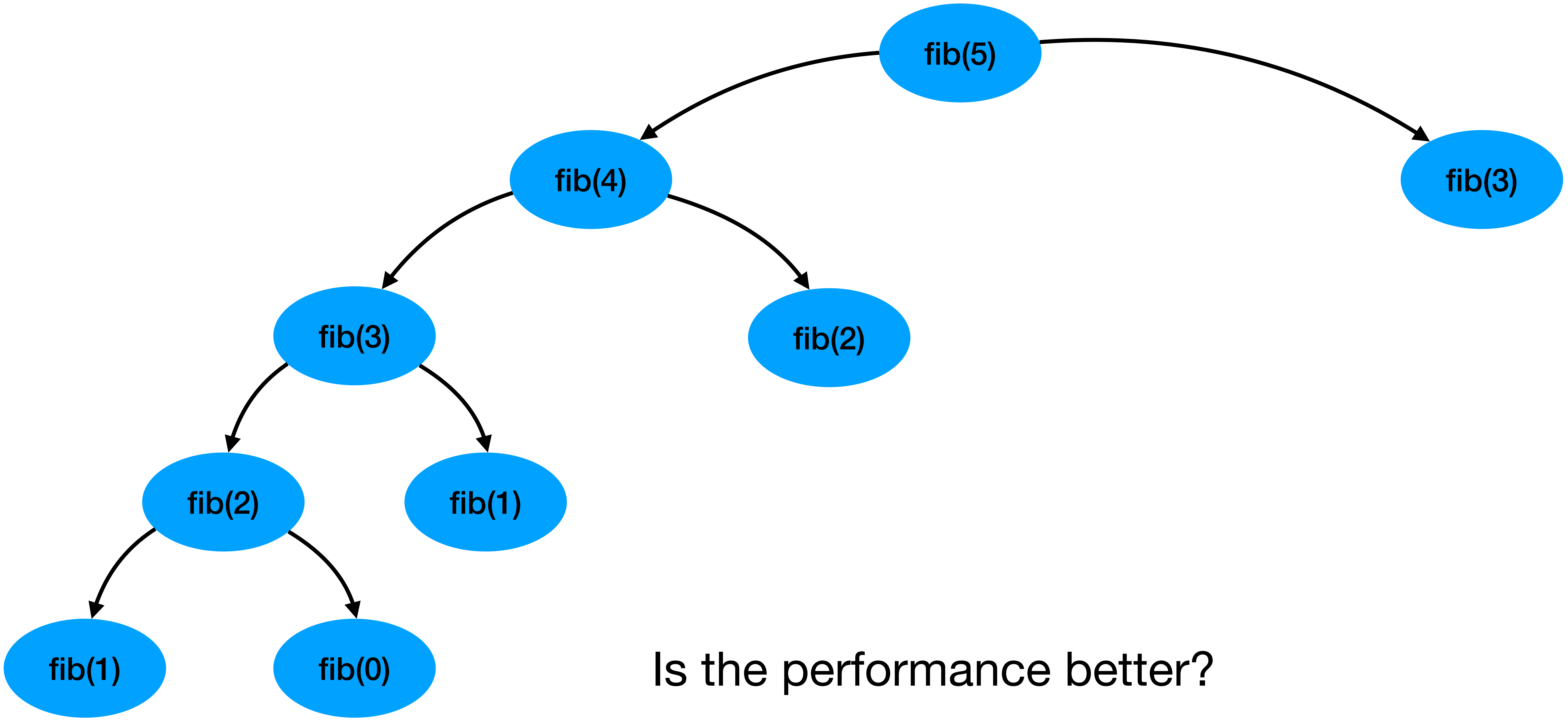
Try fib(10), fib(20), fib(30), fib(40), fib(50)

What do you notice???

Reminder: `return fib(n - 1) + fib(n - 2);`

Example: fibonacci + memoization

Let's analyze its performance! How many times does fib(5) call itself???



Is the performance better?

Example: fibonacci + memoization

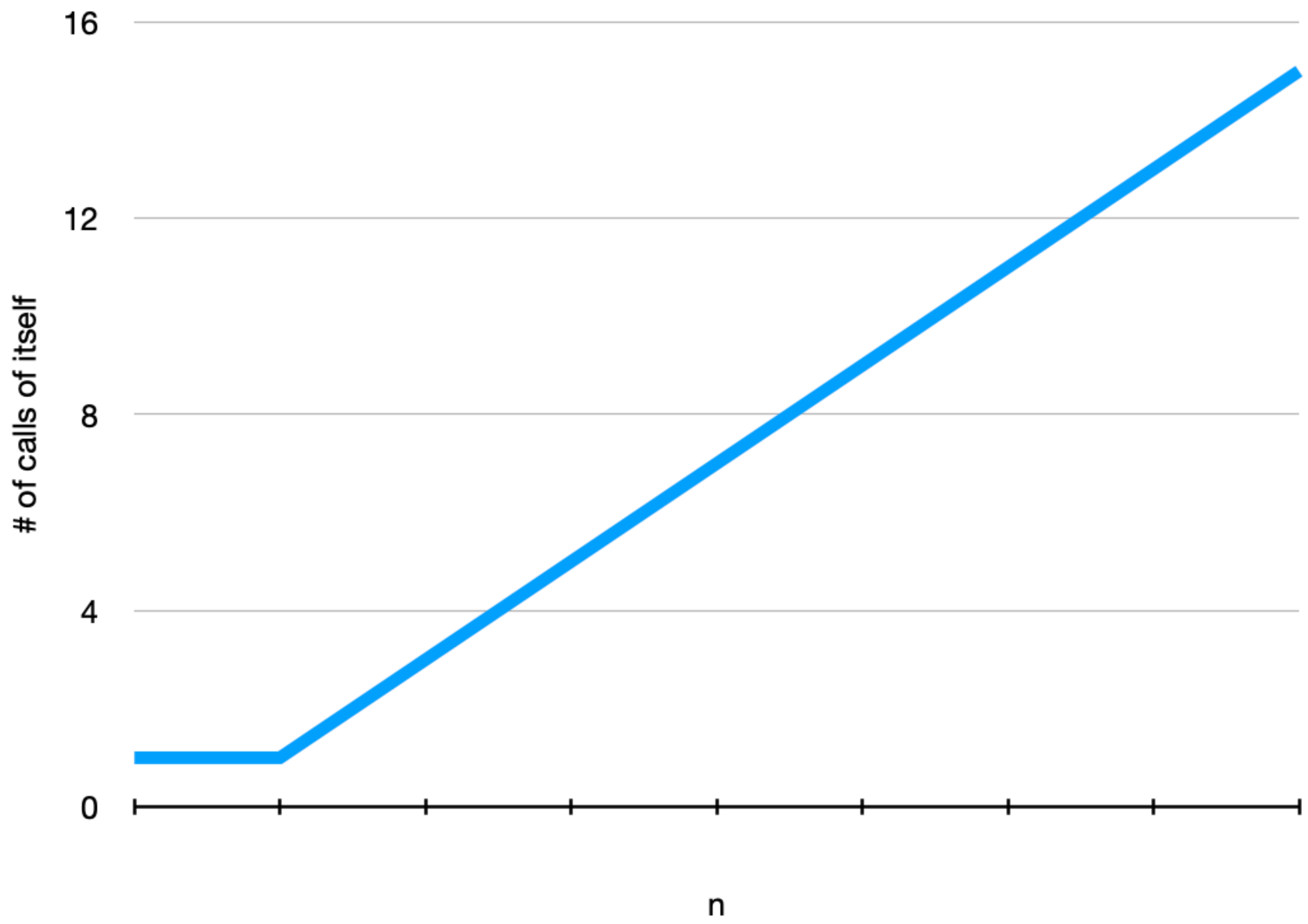
Performance analysis

- fib(4): 7 calls
- fib(5): 9 calls
- fib(6): 11 calls
- fib(7): 13 calls
- fib(8): 15 calls

… Time for a graph!

Example: fibonacci + memoization

Performance analysis



Every change in n now results

in FAR FEWER calls to fib().

IN FACT, the calls grow in a

LINEAR fashion.

Linear performance is a HUGE

improvement, and is generally ok

for MANY problems

and data sets.

So are we done…?

- For many applications of fib(), yes
- BUT… try using large values of n, like 15000

What happened???

The short answer

- Calling a function uses space in an area of memory that the computer loans your program, called the **call stack**
- When a called function returns, your program gets the call stack space back that the called function was using
- BUT… recursive functions do not start giving back their call stack space UNTIL the escape clause triggers.
    - For fib(11000), that is 11,000 function calls deep before the call stack starts to get some of its memory back. And modern call stacks are typically not large enough for that.
- This is an inherent and unavoidable problem with recursion using modern computer architecture. The only way around it is to NOT USE RECURSION.

For every recursive solution…

There is also a non-recursive solution

- Pros
  - More performant, no call stack overflows
- Cons
  - [Much] less readable, doesn't look ANYTHING like the original recursive definition

Example: fibonacci non-recursive

```
public static long fib(int n) {
    long fibN = 0;
    long nMinus1 = 1;
    long nMinus2 = 0;

    if(n == 0) {
        return 0;
    }
    if(n == 1) {
        return 1;
    }

    for(int i = 2; i <= n; i++) {
        fibN = nMinus1 + nMinus2;
        nMinus2 = nMinus1;
        nMinus1 = fibN;
    }
    return fibN;
}
```

Try fib(11000), fib(20000), fib(30000)

What do you notice???

Summary

- Recursive solutions are elegant, beautiful, and the code is easy to read
- Recursive definitions can be hard to figure out
- Recursion can kill your program
- Use them when you know HOW your program will use the recursive function
  - E.g., if you expect large data sets, you should look for a non-recursive algorithm
  - If the max number of possible recursive calls is low, recursion is ok AND consider memoization

Head Recursion

- The recursive call IS NOT the last statement in the function
- The function does its “work” AFTER the recursive call(s)
- Our fib() functions are head recursive

Tail Recursion

- The recursive call occurs as the last statement in the function
- Does its processing BEFORE the recursive call

Example: naive fibonacci

Refactored so head recursion is easier to see

```
public static long fib(int n) {
    if(n == 0) {
        return 0;
    }
    if(n == 1) {
        return 1;
    }

    // do recursive calls first
    long f1 = fib(n - 1);
    long f2 = fib(n - 2);

    // then do the work (adding)
    return f1 + f2;
}
```

Example: naive fibonacci

Tail recursion version

```
// swiped from https://www.geeksforgeeks.org/tail-recursion-fibonacci/

public static long fib(int n, long a, long b) {
    if (n == 0)
        return a;
    if (n == 1)
        return b;

    return fib(n - 1, b, a + b);
}

public static void main(String[] args) {
    System.out.println(fib(10, 0, 1));
}
```

Notice that it adds BEFORE the recursive call.

Very strange. Does this look like a version of fibonacci we did earlier?

Exercise
min()

1. Write the <u>**recursive definition**</u> of an algorithm to find the **minimum value** in an array of ints

   Express the definition as the 2 hallmarks

2. Program it

Exercise

Recursive definition of min()

- Hallmark 1: min(a, n) = smaller of a[n-1] and min(a, n-1)
- Hallmark 2: min(a, 1) = a[0]

Exercise

min()

BONUS: write the tail recursive version of min()