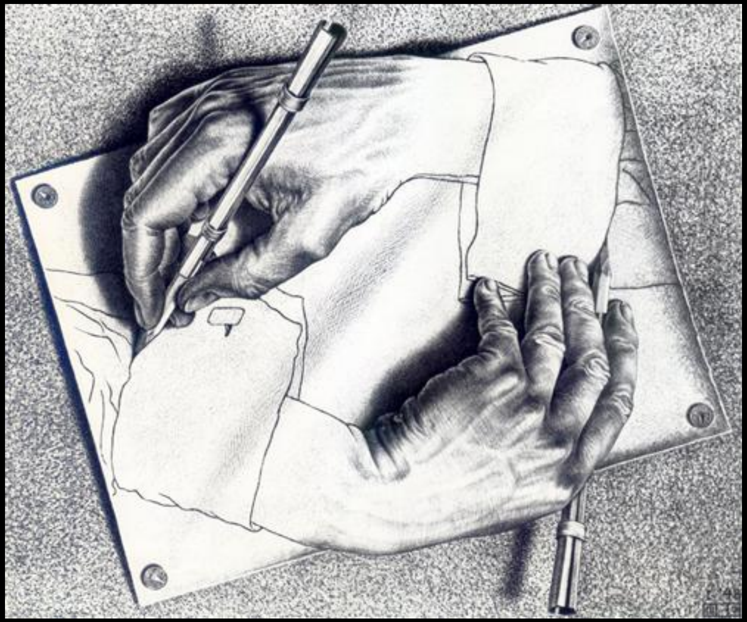
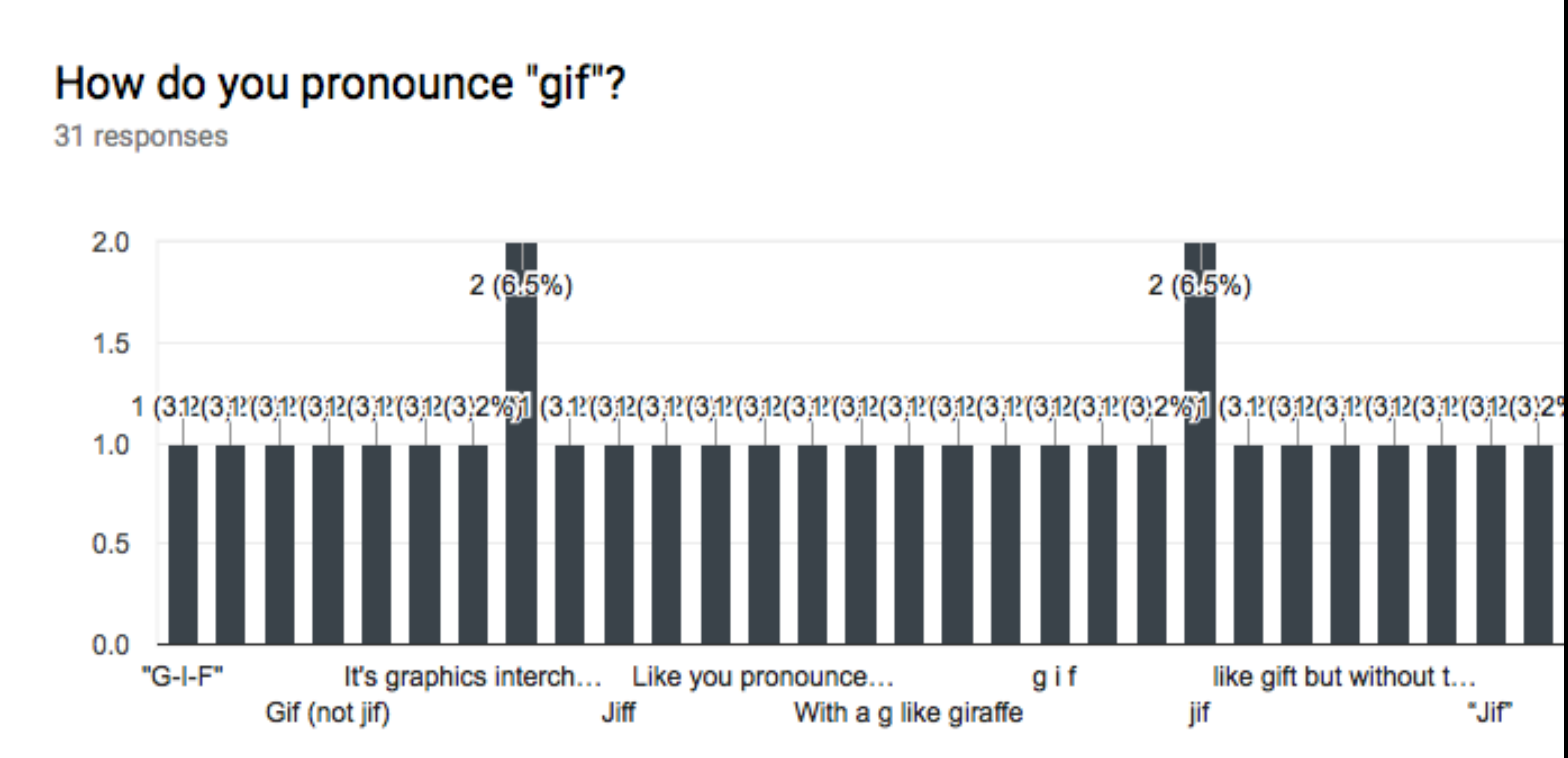
#2 (More) Environments and Recursion

TA: Jerry Chen (jerry.c@berkeley.edu)



Drawing Hands by M. C. Escher

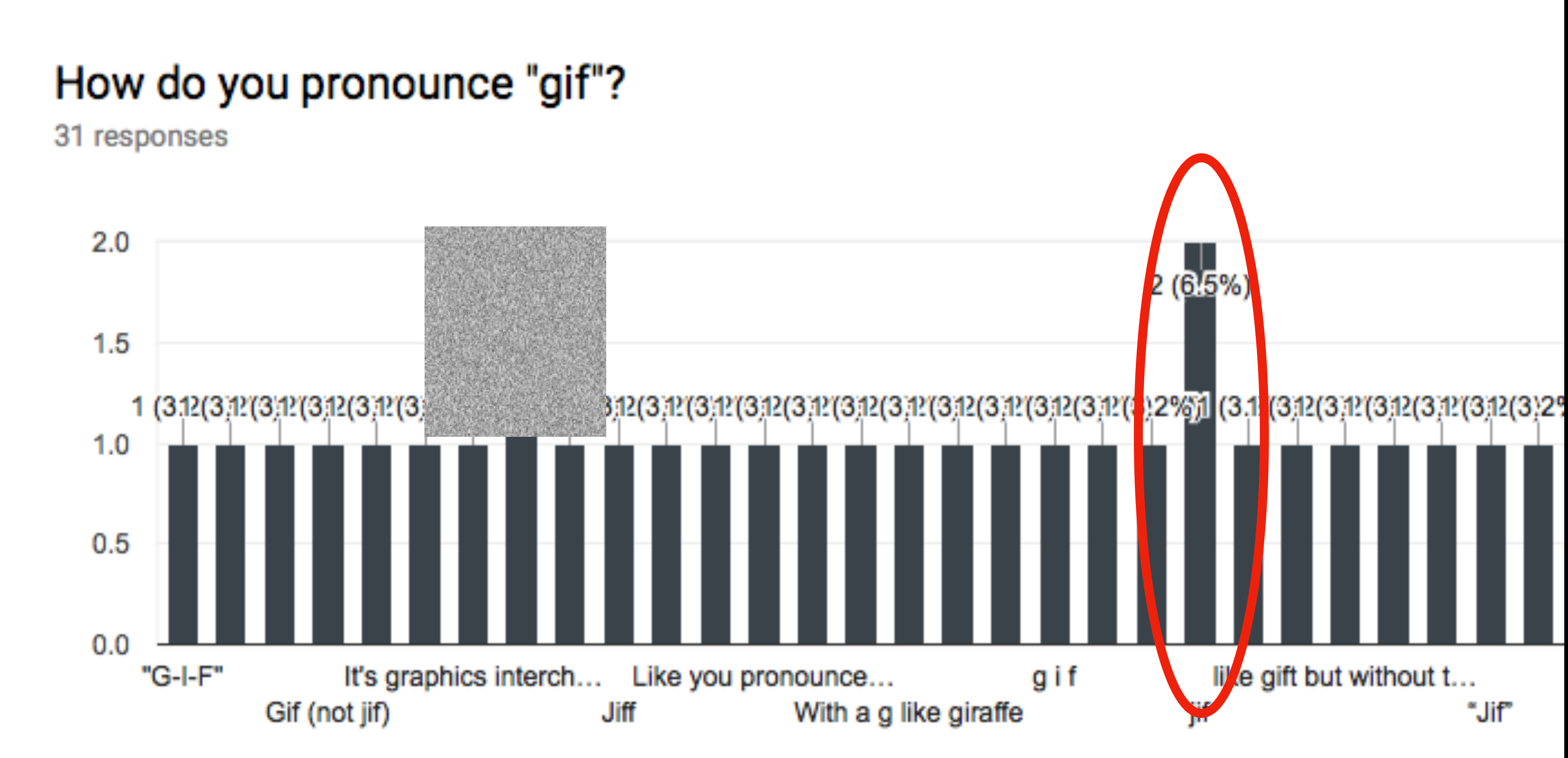
Just for Fun



How do you pronounce "gif"?

31 responses

Clear winner, "jif"!

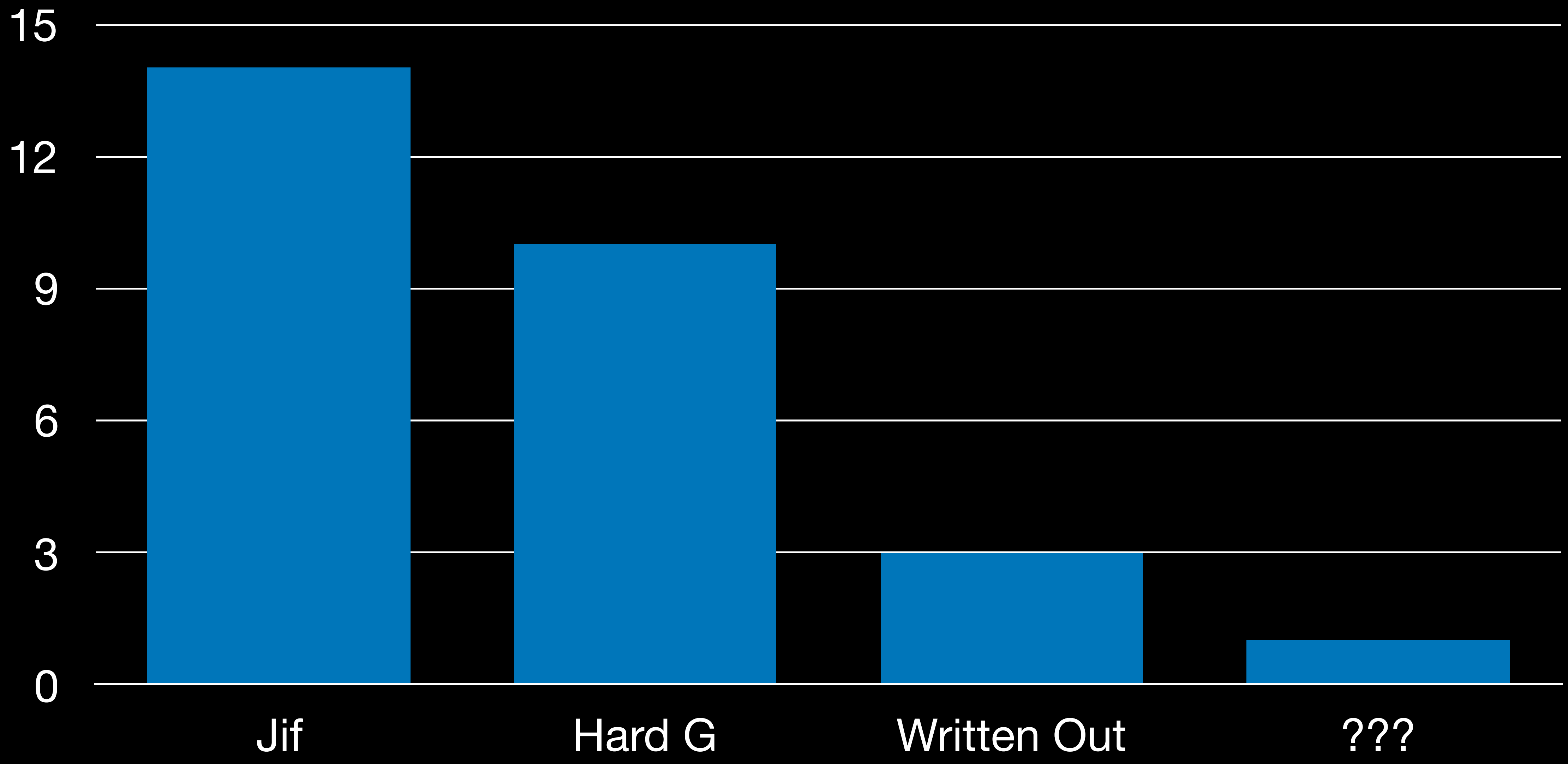


How do you pronounce "gif"?

31 responses

shh

The real tally



Code Style

Why care about code style?

- The Python interpreter doesn't really care
- You want your boss to understand your code
- You want your coworkers to understand your code
- You want **future you** to understand your code

Composition

Two main parts

- Syntactical quibbles
- Content choice and structure

Composition

Two main parts (for an English essay)

- Grammar and spelling
- Content choice and structure

Composition

- https://cs61a.org/articles/composition.html
- Syntax is easy to check: http://flake8.pycqa.org/en/latest/
- Content requires more human effort

Composition

A few big ideas

- The "best" code is <u>self-explanatory</u>
- Remove <u>repetition</u> and don't repeat yourself
- Reduce <u>length</u> without compromising readability

Writing "Self-Explanatory" Code

```python
# If x is in range and x is even then return True
if x>10 and x<100 and x%2 == 0:
    return True
else:
    return False
```

Writing "Self-Explanatory" Code

```python
# If x is in range and x is even then return True
if x>10 and x<100 and x%2 == 0:
    return True
else:
    return False
```

```python
in_range = lambda x: x>10 and x<100
is_even = lambda x: x%2 == 0

if in_range(x) and is_even(x):
    return True
return False
```

Is the earlier comment necessary?

Repetition

```
while x < max_val:
    if x % 2 == 0:
        handle_a(x)
        x += 1
    else:
        handle_b(x)
        x += 1
```

Repetition

```python
while x < max_val:
    if x % 2 == 0:
        handle_a(x)
        x += 1
    else:
        handle_b(x)
        x += 1
```

```python
while x < max_val:
    if x % 2 == 0:
        handle = handle_a
    else:
        handle = handle_b
    handle(x)
    x += 1
```

Repetition

Bonus: reduce nesting and length of loop code

```
while x < max_val:
    if x % 2 == 0:
        handle_a(x)
        x += 1
    else:
        handle_b(x)
        x += 1
```

```
def choose_handle(x):
    ...

while x < max_val:
    handle = choose_handle(x)
    handle(x)
    x += 1
```

Even if the overall code is longer, the while clause is shorter and easier to read

Length and readability

Sometimes you bark up the wrong tree

```python
def double_eights(n):
    prev_eight = False
    while n > 0:
        last_digit = n % 10
        if last_digit == 8 and prev_eight:
            return True
        elif last_digit == 8:
            prev_eight = True
        else:
            prev_eight = False
        n = n // 10
    return False
```

Length and readability

Sometimes you bark up the wrong tree

```python
def double_eights(n):
    while n > 10:
        if n % 100 == 88:
            return True
        n = n // 10
    return False
```

Bonus*

Sometimes, that tree is shorter than you think‡

```python
def double_eights(n):
    return '88' in str(n)
```

*(You haven't learned this in class yet)

‡(Yeah, it's a weird analogy)

Composition

In Conclusion

- There rarely is a "best" way
- The "best" way is even more rarely obvious
- All good code has its genesis in bad code

Environment Diagrams

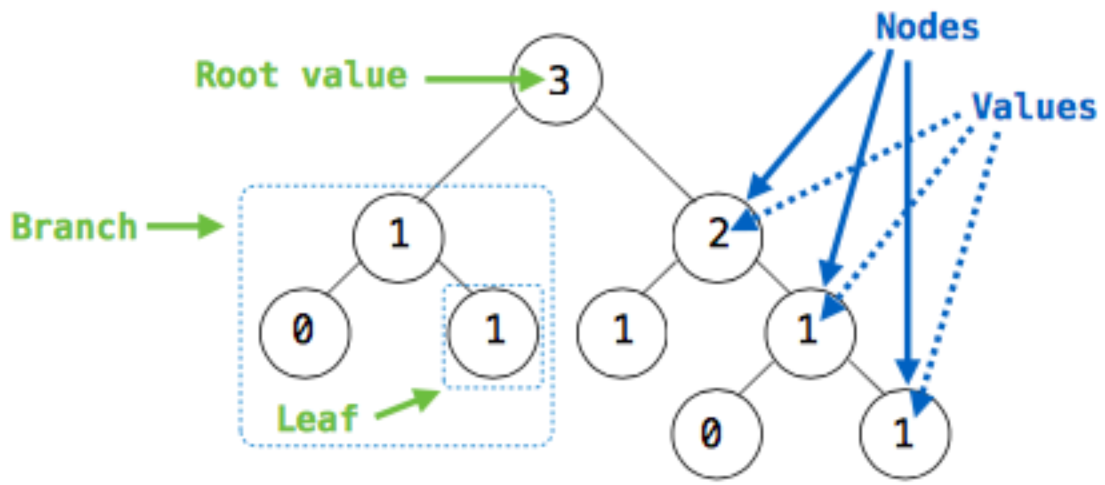
Environment Diagram Rules

- **Names** can also be bound to functions!
- **Function call:** create and number new frame (f1, f2, etc.)

  — always start in global frame
- **Assignment:** write variable name and expression value
- **Def statements:** record function name and bind function object. Remember parent frame!
- Frames **return values** upon completion (Global is special)

Recursion

Tree Abstraction



Recursive description **(wooden trees):**

A **tree** has a **root** value and a list of **branches**

Each branch is a **tree**

A tree with zero branches is called a **leaf**

Relative description **(family trees):**

Each location in a tree is called a **node**

Each **node** has a **value**

One node can be the **parent/child** of another

People often refer to values by their locations: "each parent is the sum of its children"

Components of Recursion

3 Easy Steps

1. Solve **base case**
2. **Recursive call** on a subproblem
3. **Use the result** to solve the original problem

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

```python
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

```python
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

Base case

Recursive call

Using the result

```python
def hailstone(n):
    print(n)
    if n == 1:
        return
    elif n % 2 == 0:
        hailstone(n - 1)
    else:
        hailstone(n - 1)
```

What's wrong?

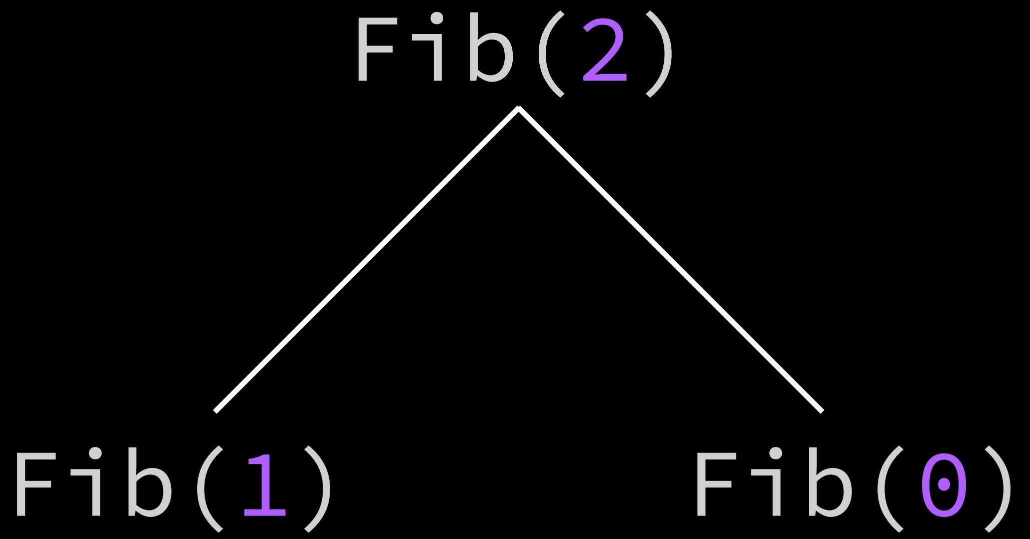
```python
def hailstone(n):
    print(n)
    if n == 1:
        return
    elif n % 2 == 0:
        hailstone(n - 1)
    else:
        hailstone(n - 1)
```

Tree Recursion

Call **multiple** functions

Useful for representing choices

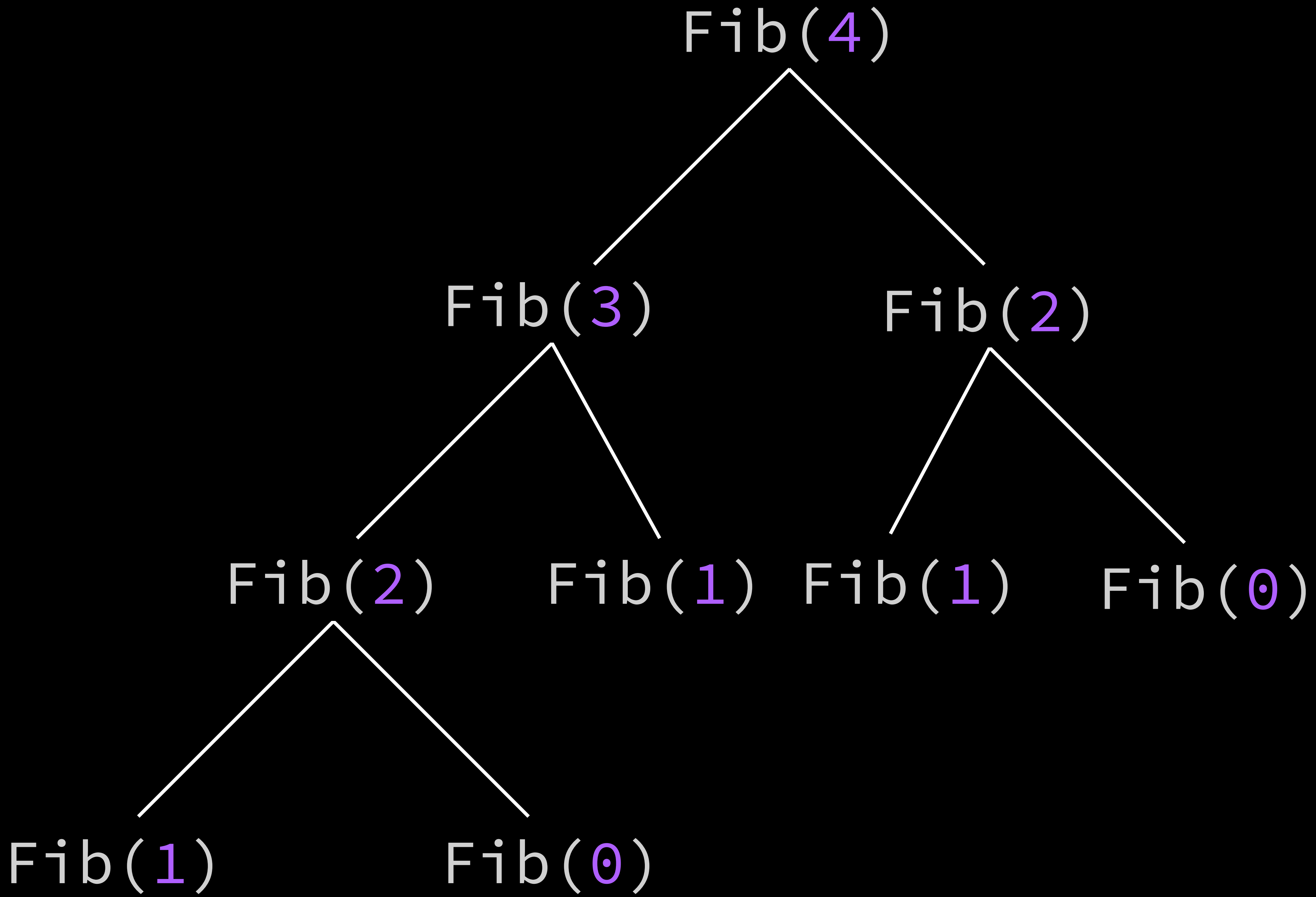
```
Fib(n) = Fib(n - 1) + Fib(n - 2)
Fib(2) = Fib(1) + Fib(0)
```



Fib(2)

Fib(1)

Fib(0)



Fib(4)

Fib(3) Fib(2)

Fib(2) Fib(1) Fib(1) Fib(0)

Fib(1) Fib(0)