Recursive Algorithms

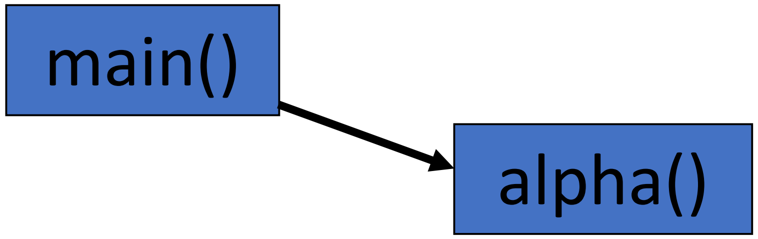
Part 2 – Python version

Roadmap

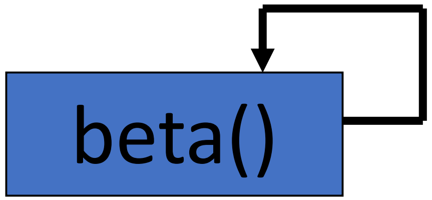
- Review of recursion
- Sample recursive algorithms
  - Factorials
  - Greatest common divisor
  - Fibonacci series

Review of recursion

- "Normally", procedures (or methods) call other procedures
  - E.g. the main() procedure calls the alpha() procedure



- A recursive procedure is one which calls itself
  - E.g. the beta() procedure contains a call to beta()



Rules for recursive algorithms

1. **<u>Base case</u>**: a recursive algorithm must always have a base case which can be solved without recursion. Methods without a base case will result in infinite recursion when run.
2. **<u>Making progress</u>**: for cases that are to be solved recursively, the next recursive call must be a case that makes progress towards the base case. Methods that do not make progress towards the base case will result in circular recursion when run.
3. **<u>Design rule</u>**: Assume that all the recursive calls work.
4. **<u>Compound interest rule</u>**: Never duplicate work by solving the same instance of a problem in separate recursive calls.

Factorials

- The factorial of a non-negative integer n may be computed as the product of all positive integers which are less than or equal to n
- This is denoted by $n!$
- In general: $n! = n \times (n-1) \times (n-2) \times (n-3) \times \ldots \times 1$
- The above is essentially an algorithm which may be implemented and used to calculate the factorial for any $n > 0$
- Note: the value of 0! is defined as 1 (i.e. 0! = 1 following the empty product convention). The input n=0 will serve as the base case in our recursive implementation.

Factorials

- Factorial operations are commonly used in many areas of mathematics, e.g. combinatorics, algebra, computation of functions such as sin and cos, and the binomial theorem.
- One of its most basic occurrences is the fact that there are $n!$ ways to arrange n distinct objects into a sequence
- In general: $n! = n \times (n-1) \times (n-2) \times (n-3) \times \ldots \times 1$
- Example factorial calculation: $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

Computing a factorial

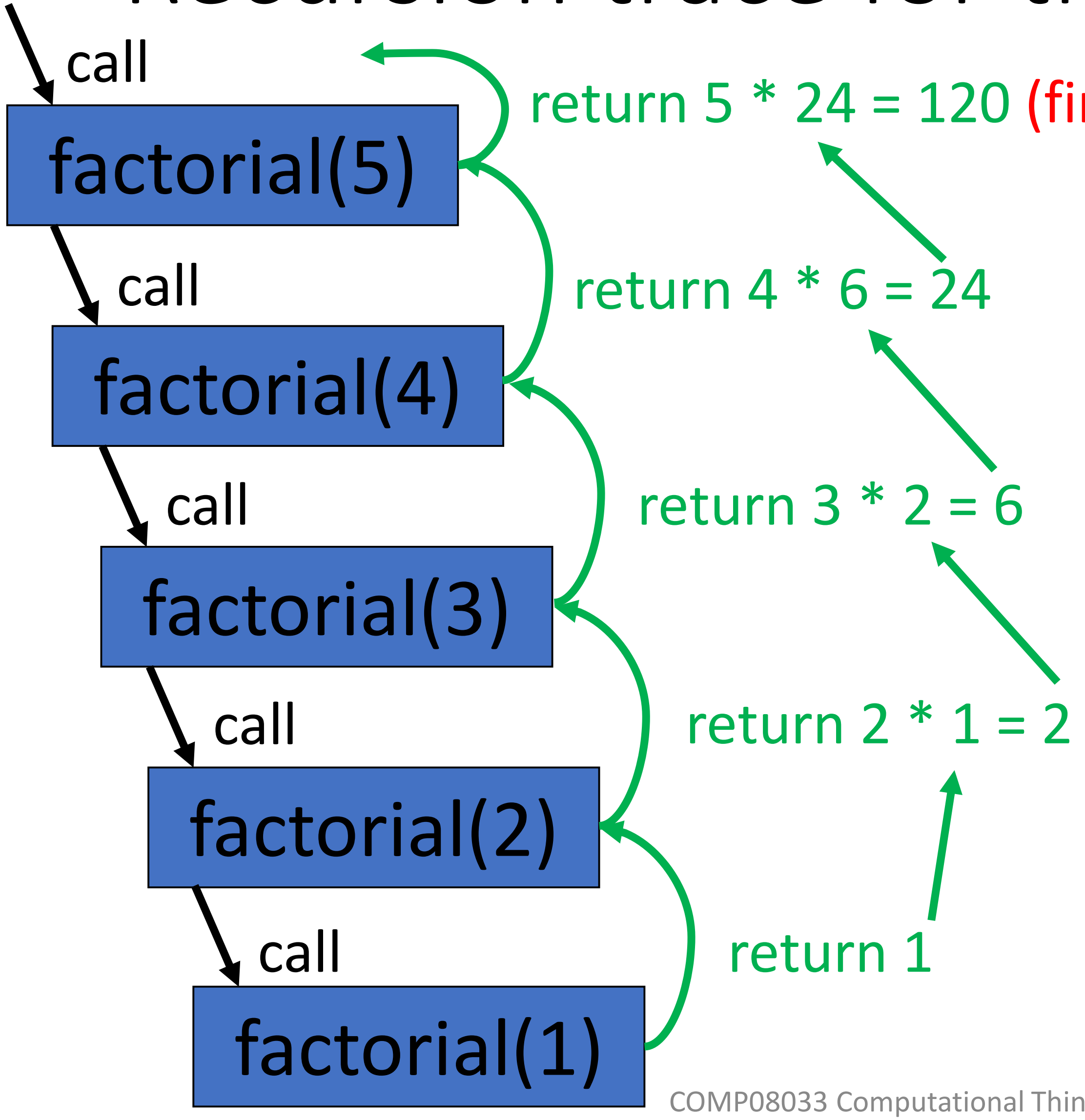
Iterative implementation

```
def factorial(n):
    answer = 1
    while n > 1:
        answer *= n
        n -= 1
    return answer


print(factorial(5))
# prints 120
```

Recursive implementation

```
def factorial_rec(n):
  if n<=1:
    return 1
  else:
    return n*factorial_rec(n-1)


print(factorial_rec(5))
# prints 120
```

Recursion trace for the call factorial(5)



Recursive implementation

```python
def factorial_rec(n):
  if n<=1:
    return 1
  else:
    return n*factorial_rec(n-1)


print(factorial_rec(5))
# prints 120
```

Greatest common divisor

- The greatest common divisor (gcd) of two integers is the largest positive integer which divides into both numbers without leaving a remainder
- E.g. the gcd of 30 and 35 is 5
- Euclid’s algorithm (c. 300 BC) may be used to determine the gcd of two integers
- Example application: finding the largest square tile which can be used to cover the floor of a room without using partial/cut tiles

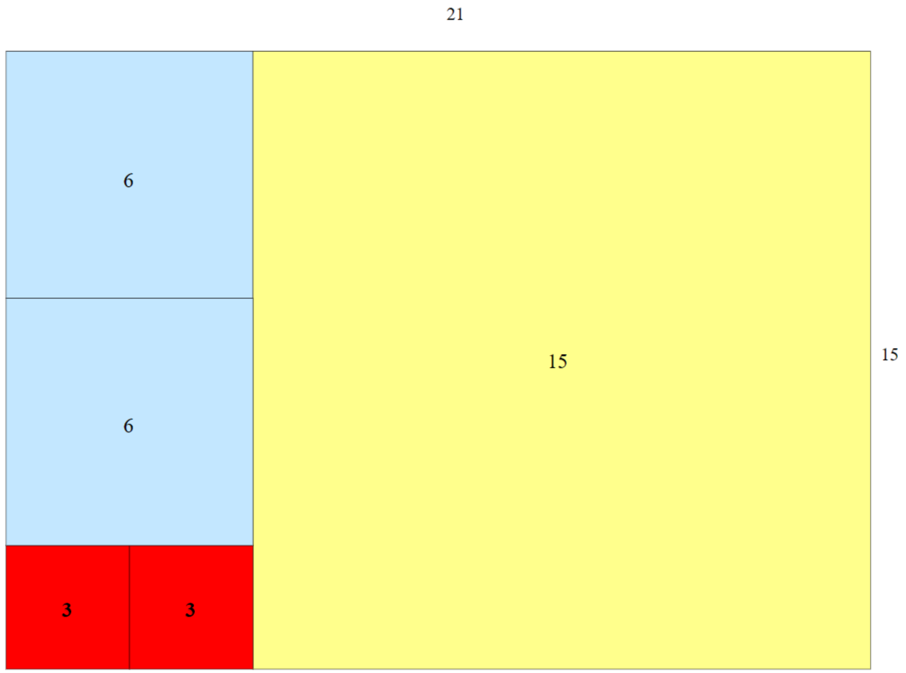


Image source: Kelam
https://commons.wikimedia.org/wiki/File:Euclide2115.svg

Computing the greatest common divisor

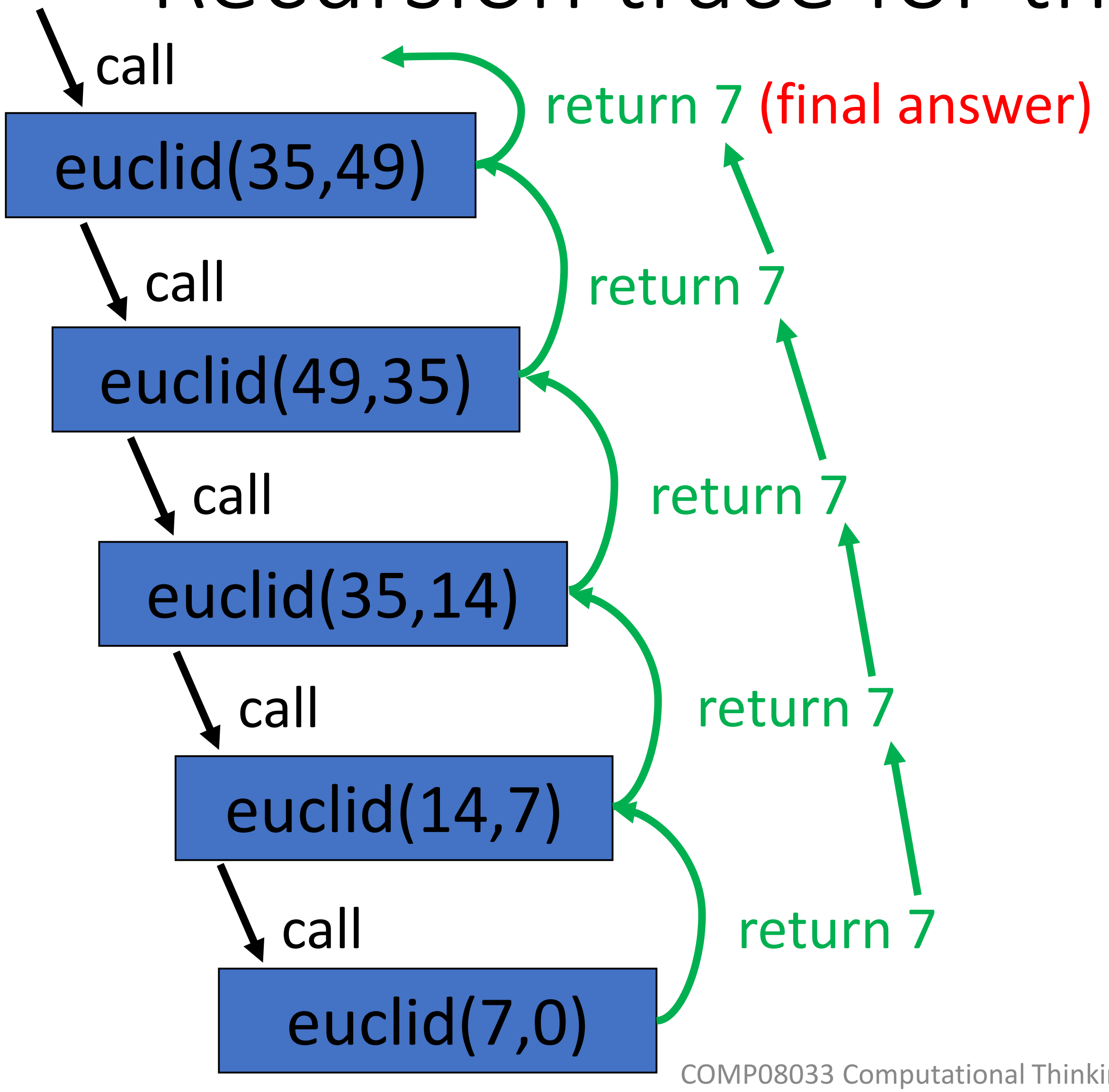
Iterative implementation

```
def euclid(a, b):
  while b != 0:
    temp = b
    b = a % b
    a = temp
  return a

print(euclid(35,49))
# prints 7
```

Recursive implementation

```
def euclid(a, b):
  if b==0:
    return a
  else:
    return euclid(b, a%b)

print(euclid(35,49))
# prints 7
```

Recursion trace for the call euclid(35,49)

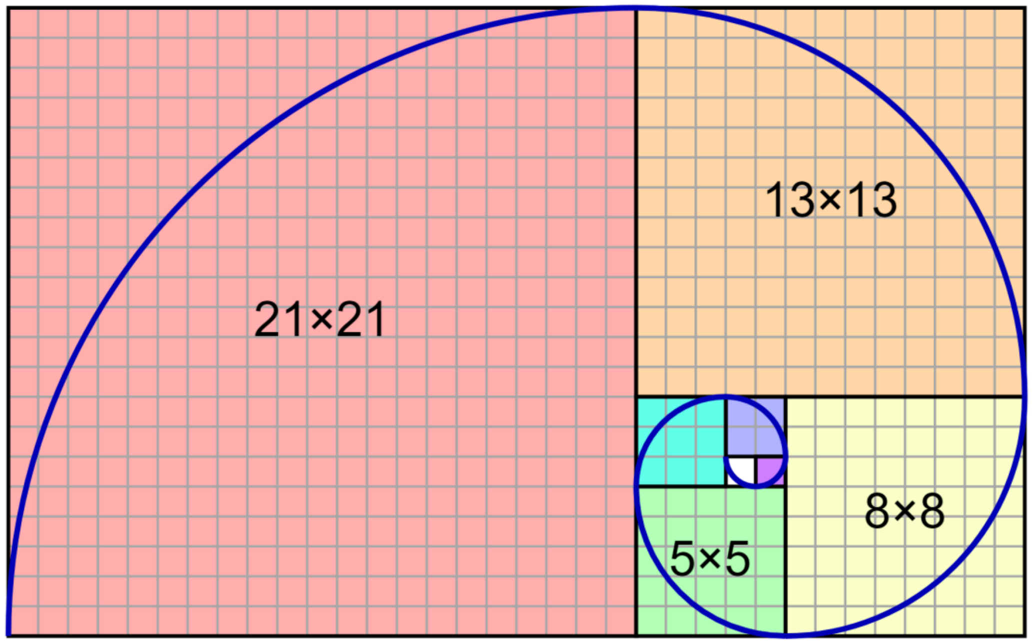


Recursive implementation

```
def euclid(a, b):
  if b==0:
    return a
  else:
    return euclid(b, a%b)

print(euclid(35,49))
# prints 7
```

The Fibonacci series

- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, …
- The Fibonacci series crops up very often in nature, and can be used to model the growth rate of organisms.
- E.g. leaf arrangement in plants, number of petals on a flower, the bracts of a pinecone, the scales of a pineapple, shells, proportions of the human body



The Fibonacci series

- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, …
- The Fibonacci series is named after the Italian mathematician Leonardo of Pisa, who was also known as Fibonacci.
- His book *Liber Abaci* (published 1202) introduced the sequence to the western world (Indian mathematicians knew about this sequence previously).
- We will use the convention that zero is included in the series and assigned to index 1
- If fib(n) is a method that returns the n^{th} number in the series, then: fib(1)=0, fib(2)=1, fib(3)=1, fib(4)=2, fib(5)=3, fib(6)=5, fib(7)=8, etc…
- In general, fib(n) = fib(n-1) + fib(n-2)
- The results for fib(1) and fib(2) do not conform to this rule; therefore they will serve as base cases in our recursive implementation

Computing the n^{th} Fibonacci number

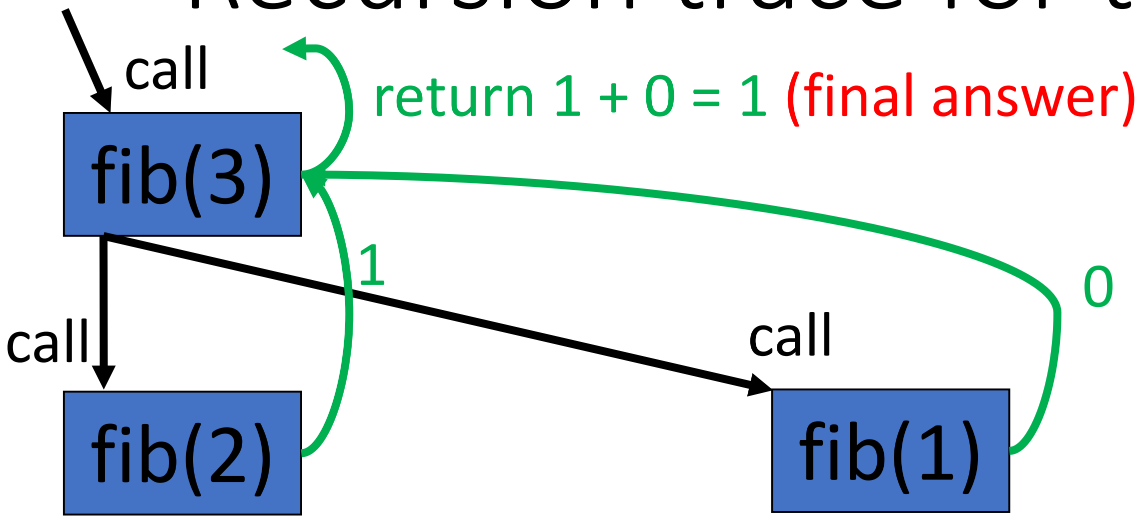
Iterative implementation

```
def fib(n):
  i, n1, n2 = 1, 0, 1
  while i < n:
    temp = n1
    n1 = n2
    n2 = n1 + temp
    i += 1
  return n1

print(fib(5))  # prints 3
```

Recursive implementation

```
def fib(n):
  if n == 1:
    return 0
  elif n == 2:
    return 1
  return fib(n-1) + fib(n-2)

print(fib(5)) # prints 3
```

Recursion trace for the call fib(3)

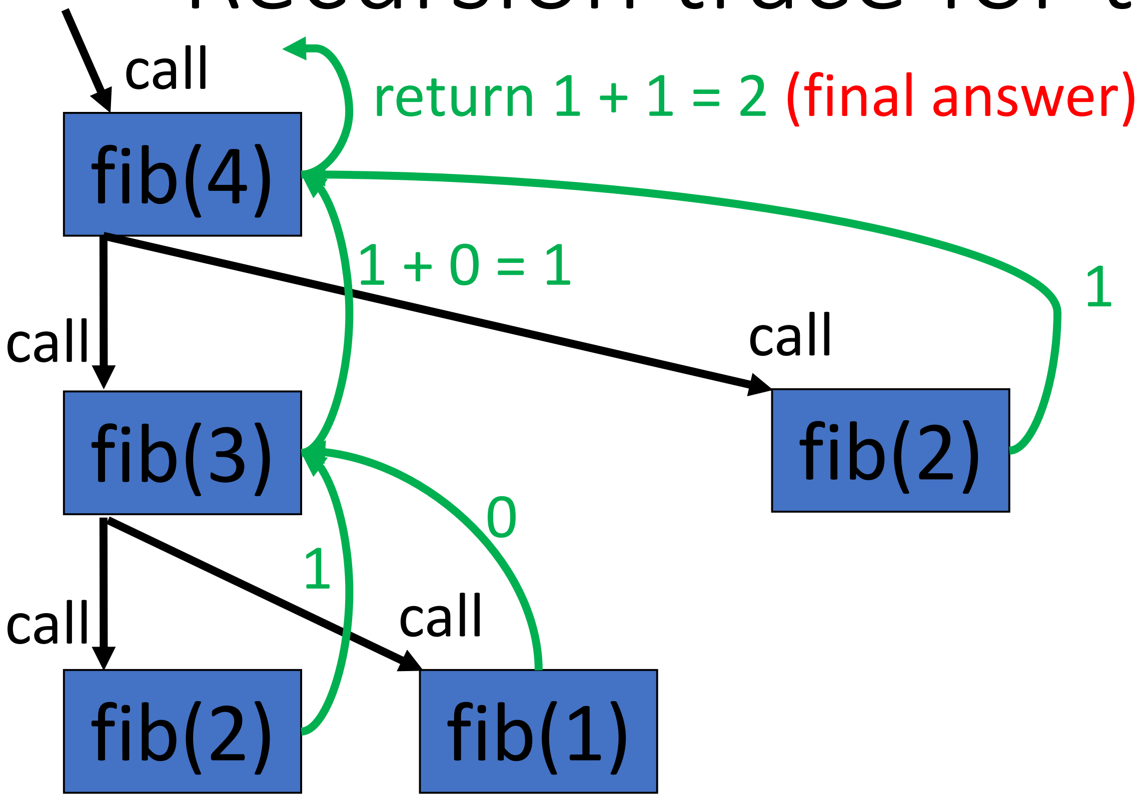


Recursive implementation

```python
def fib(n):
    if n == 1:
        return 0
    elif n == 2:
        return 1
    return fib(n-1) + fib(n-2)


print(fib(5)) # prints 3
```

- Note: this is an example of binary recursion; fib(3) makes two recursive calls, fib(2) and fib(1)
- This implementation is technically not an example of tail recursion, as the last operation completed will be the addition of the values returned by the two recursive calls

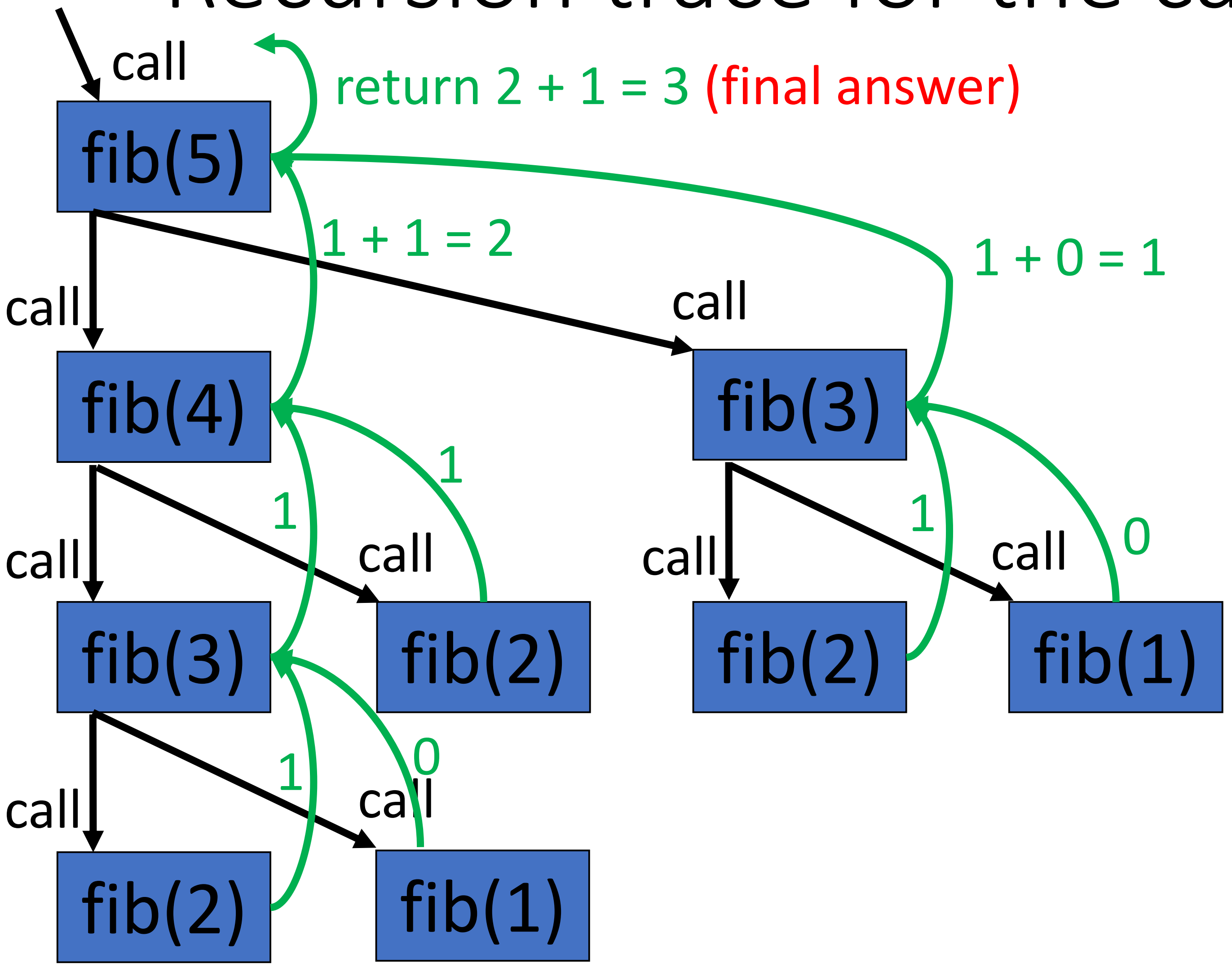
Recursion trace for the call fib(4)



Recursive implementation

```
def fib(n):
   if n == 1:
      return 0
   elif n == 2:
      return 1
   return fib(n-1) + fib(n-2)

print(fib(5)) # prints 3
```

Recursion trace for the call fib(5)



Recursive implementation

```
def fib(n):
    if n == 1:
        return 0
    elif n == 2:
        return 1
    return fib(n-1) + fib(n-2)


print(fib(5)) # prints 3
```