

CS1010S Programming Methodology

Lecture 3

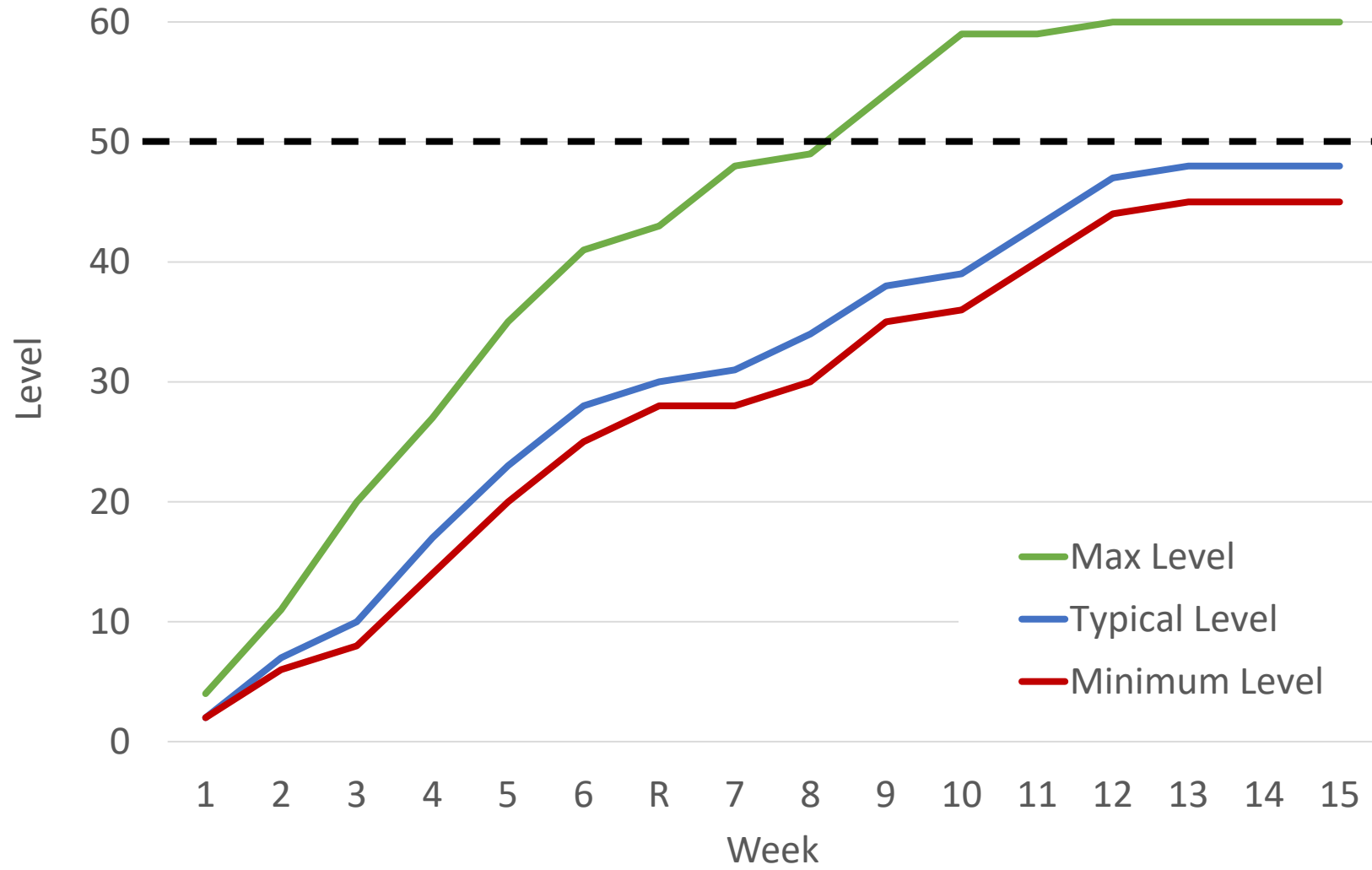
Recursion, Iteration & Order of Growth

29 Aug 2018

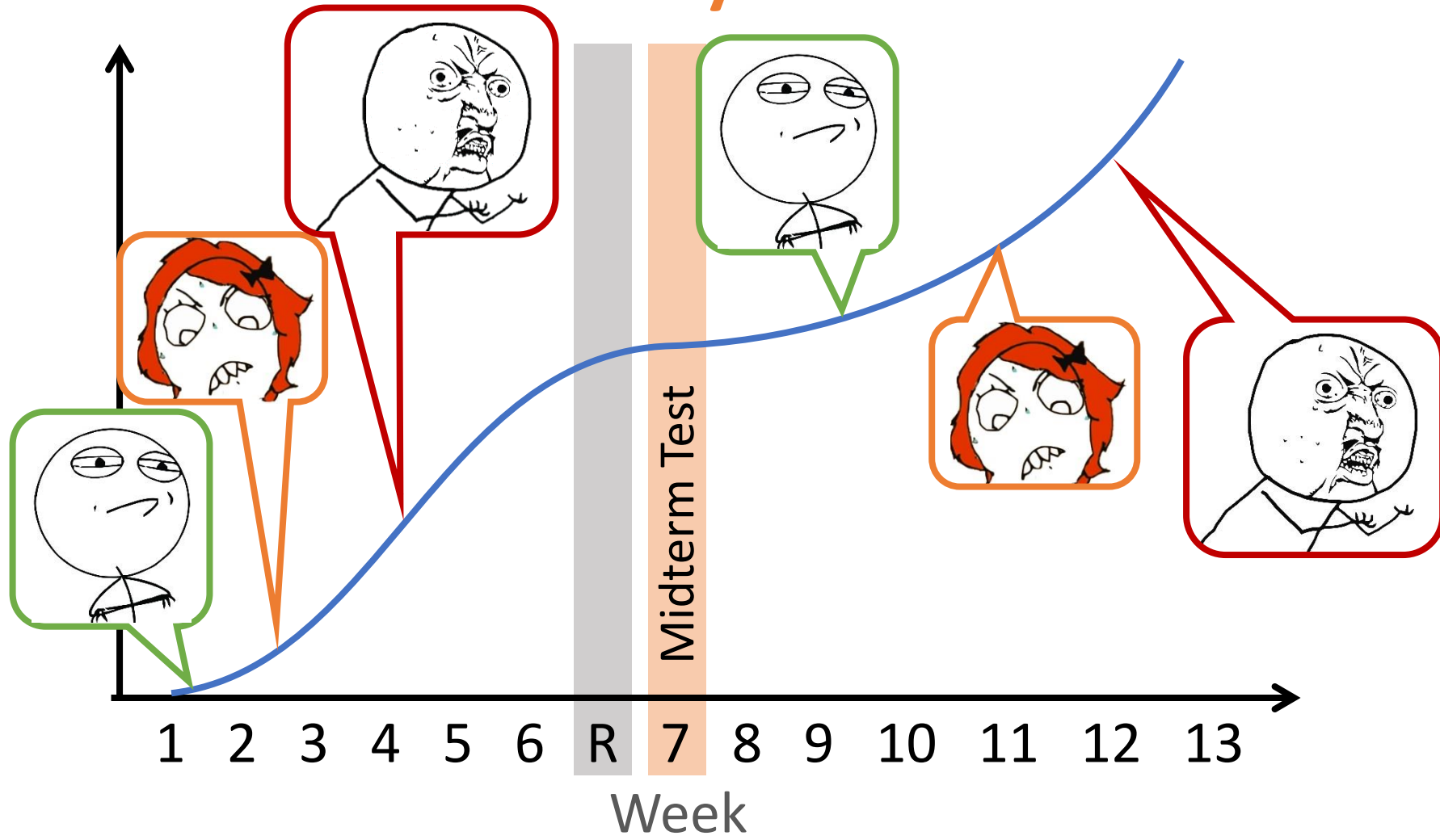
Python Problems?

cs1010s-staff@googlegroups.com

Expected Level Progression



Difficulty Curve





LEAVE NO MAN BEHIND



Reinforcements

Remedial classes

- Every week
- 6:30 – 8:30 pm
- Watch Coursemology for updates



Course Hero

Done with all the
missions?

Got a lot of time to burn?

Optional Trainings

Contests

Due 9 Sep 2018

Winning: 400 EXP + Prize

Participation: 50 EXP

Recap



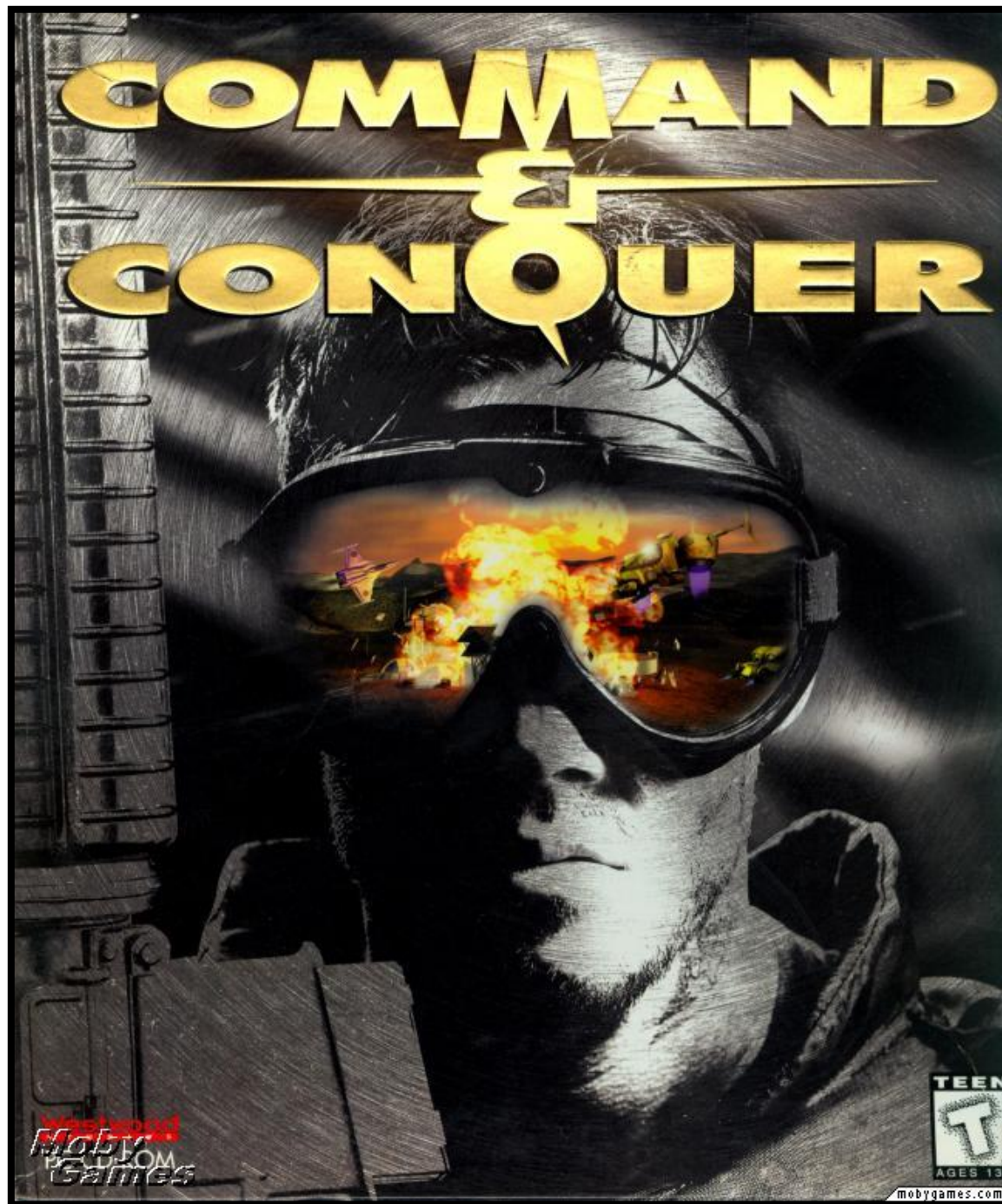
Don't need to know how it works
Just know what it does

↖ (the inputs
and output)

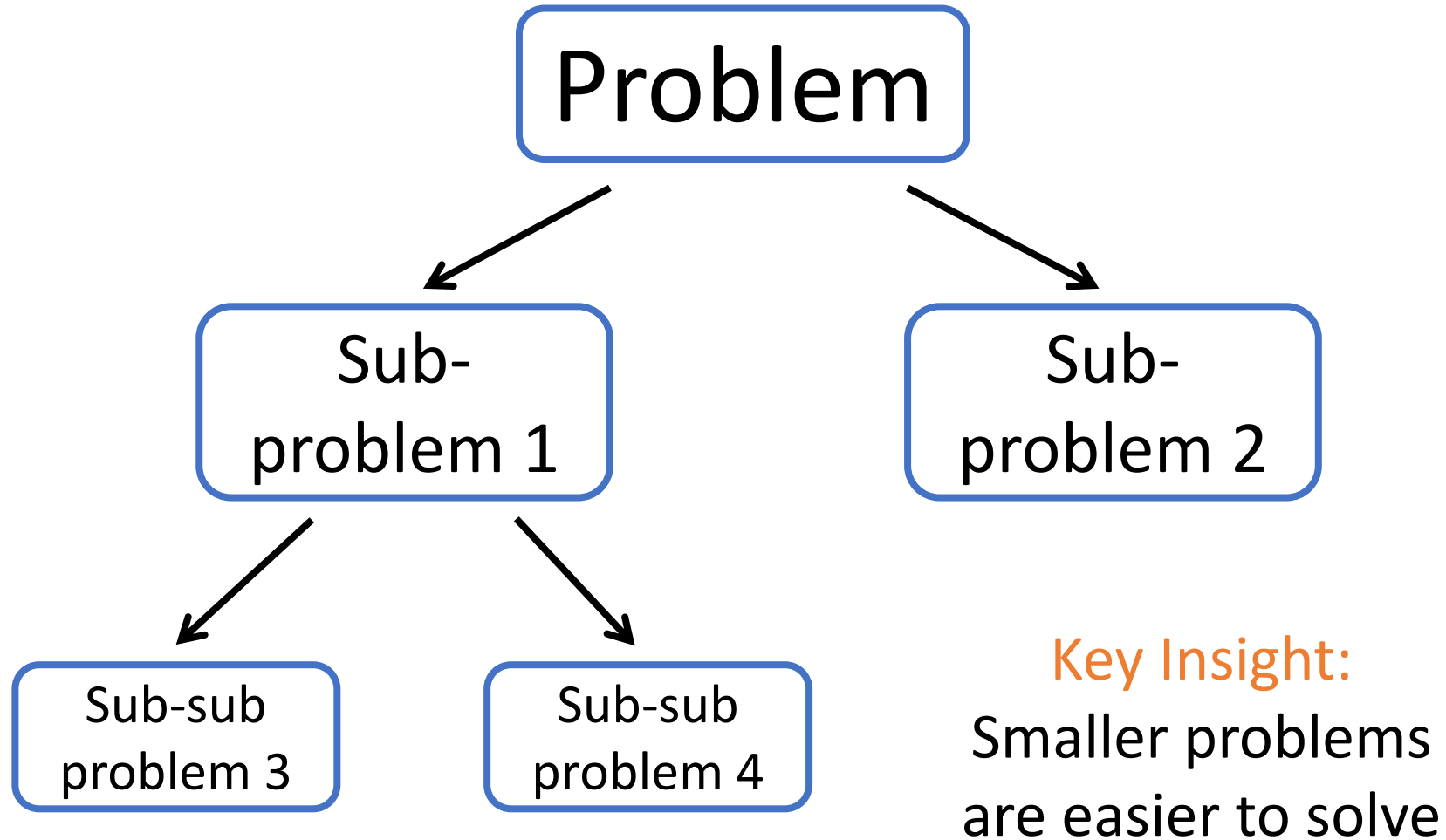
Learning Outcomes

After this lesson, you should be able to

- know how apply **divide and conquer** technique to solve a problem
- differentiate what is **recursion** and **iteration**
- state the **order of growth** in terms of **time** and **space** for computations



Divide & Conquer



What is Recursion

Smaller child problem(s) has
same structure as the parent

A recursive function is
defined as itself

e.g. $f(n) = \dots f(m) \dots$

Analogy

Your friend is late for lecture...



How to find your row?

The Strategy

- Your row number is 1 more than the row in front of you.
- Ask the person in front for his/her row number and add 1 to it.
- The person in front uses the same strategy.
- Eventually, person in front row simply replies 1.

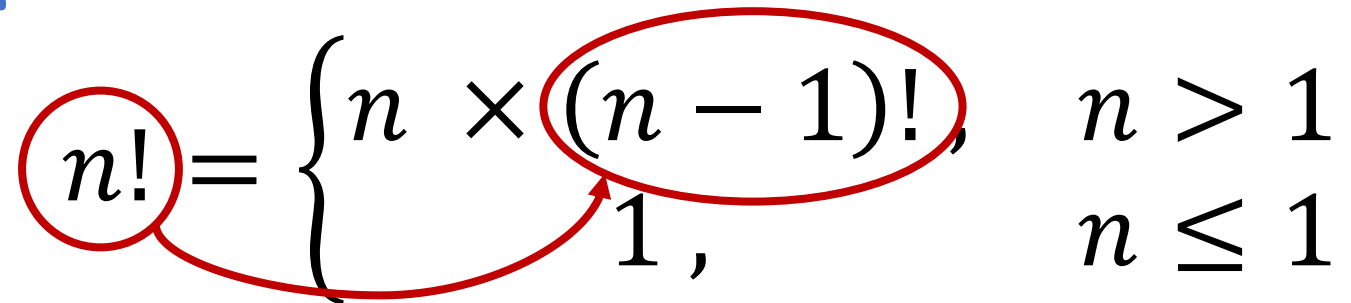
This is Recursion

Example

Consider the factorial function:

$$n! = n \times (n - 1) \times (n - 2) \cdots \times 1$$

Rewrite:



The diagram illustrates the recursive definition of the factorial function. It features the equation $n! = \begin{cases} n \times (n-1)!, & n > 1 \\ 1, & n \leq 1 \end{cases}$. Red annotations highlight the recursive structure: a red circle around $n!$ on the left, a red oval around $(n-1)!$ in the first case, and a red arrow pointing from the $n!$ circle to the $(n-1)!$ oval, indicating the recursive call.

$$n! = \begin{cases} n \times (n - 1)!, & n > 1 \\ 1, & n \leq 1 \end{cases}$$

Factorial

$$n! = \begin{cases} n \times (n - 1)!, & n > 1 \\ 1, & n \leq 1 \end{cases}$$

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

Recursion

```
def factorial(n):  
    if n <= 1: } terminating  
        return 1 } condition  
    else:  
        return n * factorial(n - 1)  
                                recursive call
```

Function that calls itself is called a **recursive** function

Recursive process

factorial(5)

5 * factorial(4)

5 * (4 * factorial(3))

5 * (4 * (3 * factorial(2)))

5 * (4 * (3 * (2 * factorial(1))))

5 * (4 * (3 * (2 * 1)))

5 * (4 * (3 * 2))

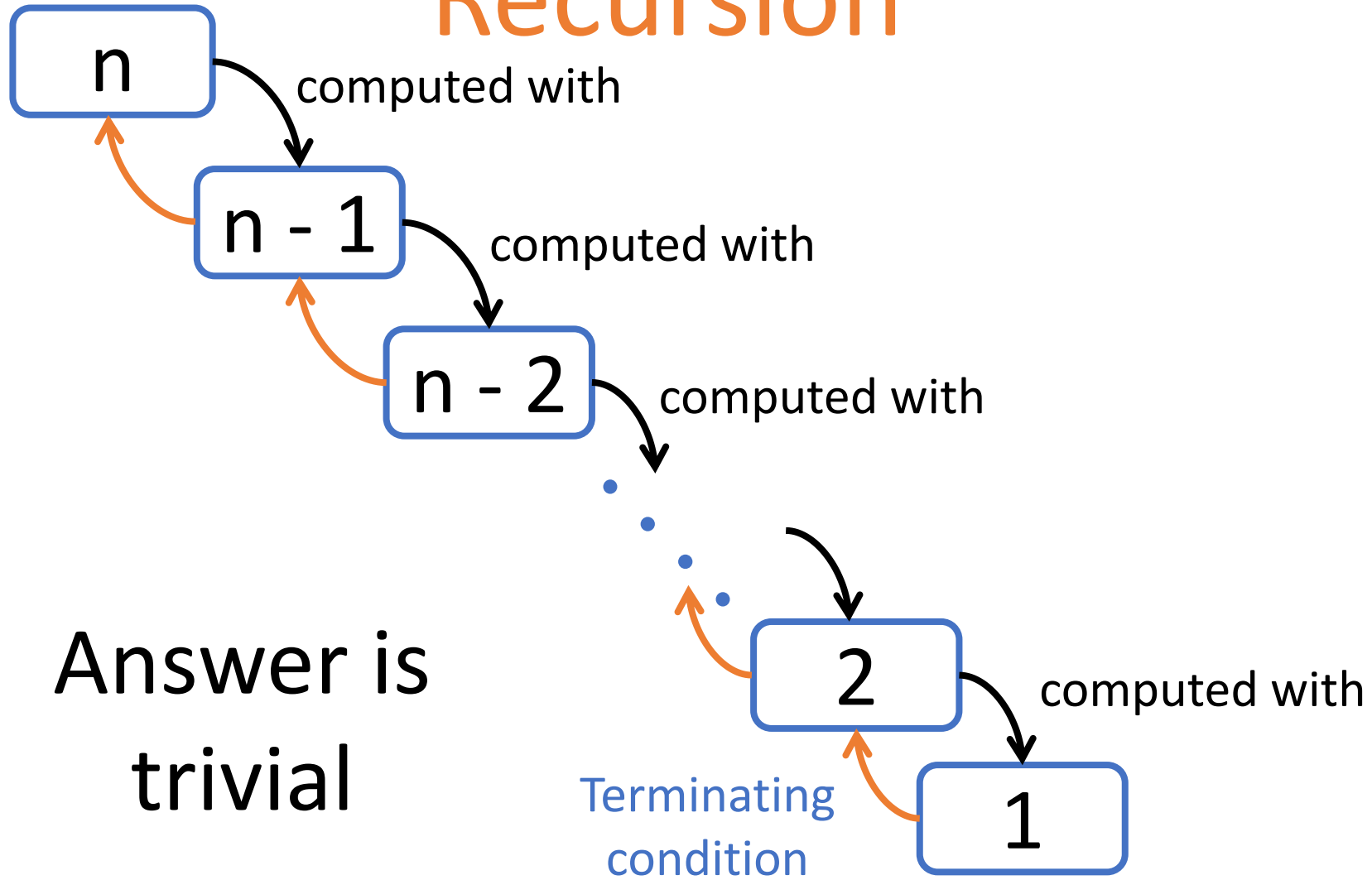
5 * (4 * 6)

5 * 24

120

Note the build up of pending operations.

Recursion



How to write recursion

1. Figure out the **base case**
 - Typically $n = 0$ or $n = 1$
2. Assume you know how to solve $n - 1$
 - Now how to solve for n ?

Factorial: Linear recursion

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

factorial(4)
↓
factorial(3)
↓
factorial(2)
↓
factorial(1)



Fibonacci Numbers

Leonardo Pisano Fibonacci (12th century) is credited for the sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Note: each number is the sum of the previous two.

Fibonacci in Math

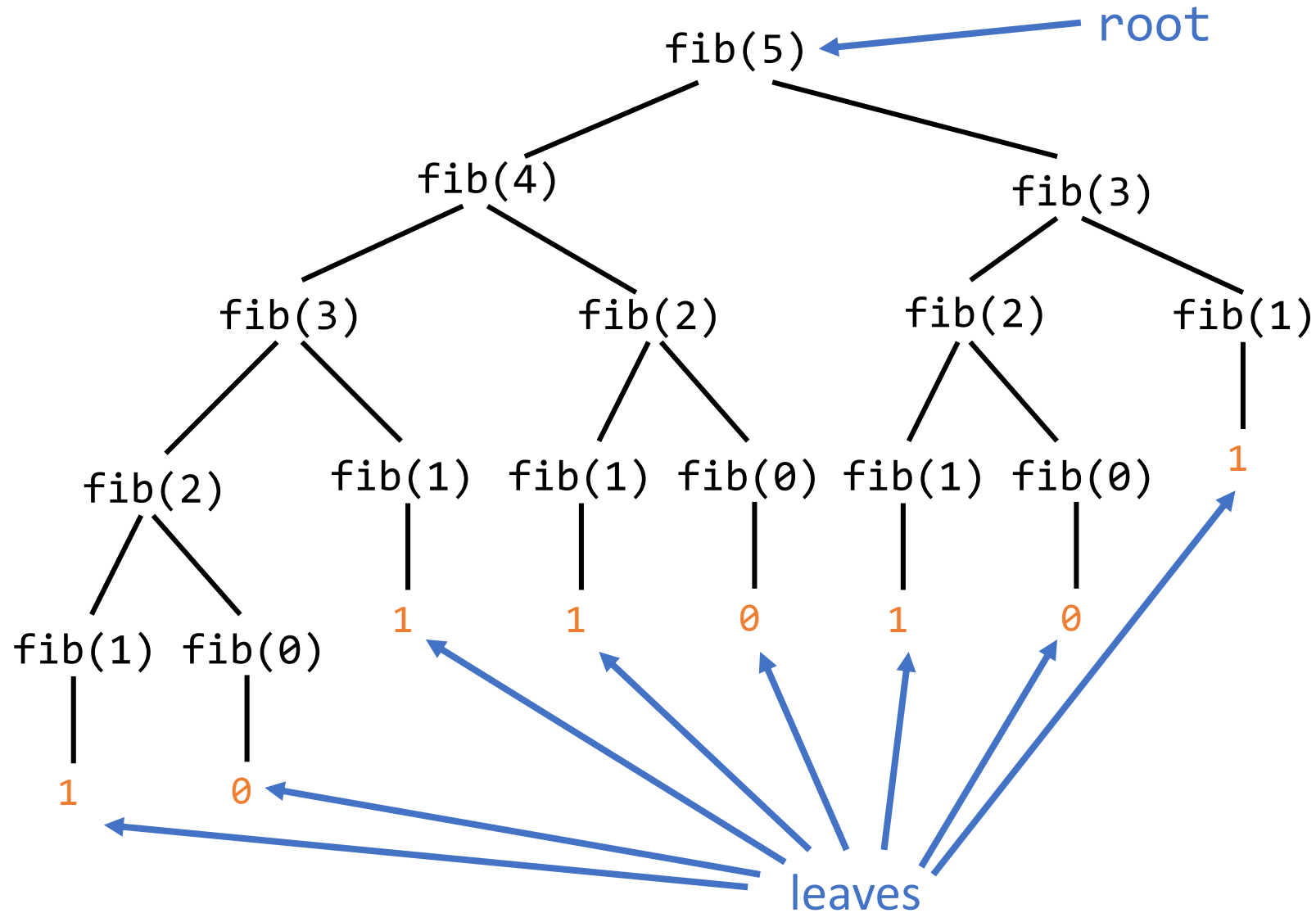
$$fib(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ fib(n-1) + fib(n-2) & n > 1 \end{cases}$$

Fibonacci in Python

$$fib(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ fib(n - 1) + fib(n - 2) & n > 1 \end{cases}$$

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

Tree recursion



Mutual recursion

```
def ping(n):  
    if (n == 0):  
        return n  
    else:  
        print("Ping!")  
        pong(n - 1)
```

```
def pong(n):  
    if (n == 0):  
        return n  
    else:  
        print("Pong!")  
        ping(n - 1)
```

ping(10)

Ping!

Pong!

Ping!

Pong!

Ping!

Pong!

Ping!

Pong!

Ping!

Pong!

Iteration

the act of repeating a process with the aim of approaching a desired goal, target or result.

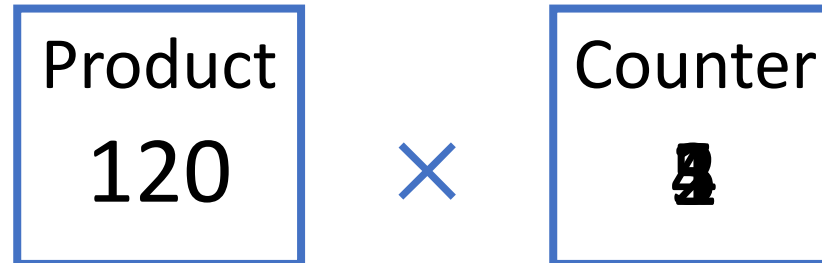
- *Wikipedia*

Iterative Factorial

Idea

Start with 1, multiply by 2, multiply by 3, ... , multiply by n.

$$n! = 1 \times 2 \times 3 \cdots \times n$$



Iterative Factorial

$$n! = 1 \times 2 \times 3 \cdots \times n$$

Computationally

Starting:

product = 1

counter = 1

Iterative (repeating) step:

product \leftarrow product \times counter

counter \leftarrow counter + 1

End:

product contains the result

Iterative Factorial

Start with 1, multiply by 2, multiply by 3, ...

$$n! = 1 \times 2 \times 3 \cdots \times n$$

Python Code

```
def factorial(n):  
    product, counter = 1, 1  
    while counter <= n:  
        product = product * counter  
        counter = counter + 1  
    return product
```

while loop

```
while <expression>:  
    <body>
```

expression

- Predicate (condition) to stay within the loop

body

- Statement(s) that will be evaluated if predicate is True

Yet another way

$$n! = 1 \times 2 \times 3 \cdots \times n$$

Factorial rule:

product \leftarrow product \times counter

counter \leftarrow counter + 1

```
def factorial(n):  
    product = 1  
    for counter in range(2, n+1):  
        product = product * counter  
    return product
```

non-inclusive.

Up to n.



for loop

```
for <var> in <sequence>:  
    <body>
```

sequence

- a sequence of values

var

- variable that take each value in the sequence

body

- statement(s) that will be evaluated for each value in the sequence

range function

```
range([start,] stop[, step])
```

creates a **sequence** of integers

- from start (inclusive) to stop (non-inclusive)
- incremented by step

Examples

```
for i in range(10):  
    print(i)
```

```
for i in range(3, 10):  
    print(i)
```

```
for i in range(3, 10, 4):  
    print(i)
```

break & continue

```
for j in range(10):  
    print(j)  
    if j == 3:  
        break  
print("done")
```

0

1

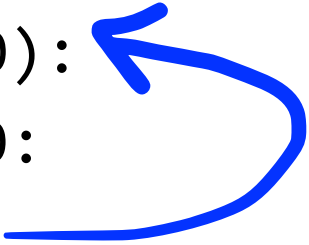
2

3

done

Break out
of loop

```
for j in range(10):  
    if j % 2 == 0:  
        continue  
    print(j)  
print("done")
```



1

3

5

7

9

done

Continue with
next value

Iterative process

```
def factorial(n):  
    product, counter = 1, 1  
    while counter <= n:  
        product = (product *  
                   counter)  
        counter = counter + 1  
    return product
```

factorial(6)

product	counter
1	1
1	2
2	3
6	4
24	5
120	6
720	7

counter > n (7 > 6)
return product (720)

Recursion vs Iteration

Recursive process occurs when there are deferred operations.

Iterative process does not have deferred operations.

Recursive Process

factorial(5)

5 * factorial(4)

5 * (4 * factorial(3))

5 * (4 * (3 * factorial(2)))

5 * (4 * (3 * (2 * factorial(1))))

5 * (4 * (3 * (2 * 1)))

5 * (4 * (3 * 2))

5 * (4 * 6)

5 * 24

120

deferred
operations



The diagram consists of four blue arrows pointing from the text 'deferred operations' to the multiplication operators in the expression '5 * (4 * (3 * (2 * 1)))' on the line above. The arrows originate from the right side of the text and point to the '*' operators between 5 and 4, 4 and 3, 3 and 2, and 2 and 1, illustrating the sequence of operations that are deferred until the recursive calls return.

Orders of Growth

Like Physicists, we care about **two**
things:

1. Space

2. Time

Rough measure of resources
used by a computational process

Space: how much memory do we need to run the program

Time: how long it takes to run a program

Order of growth

Why do we care?

We want to know how much
resource our algorithm needs

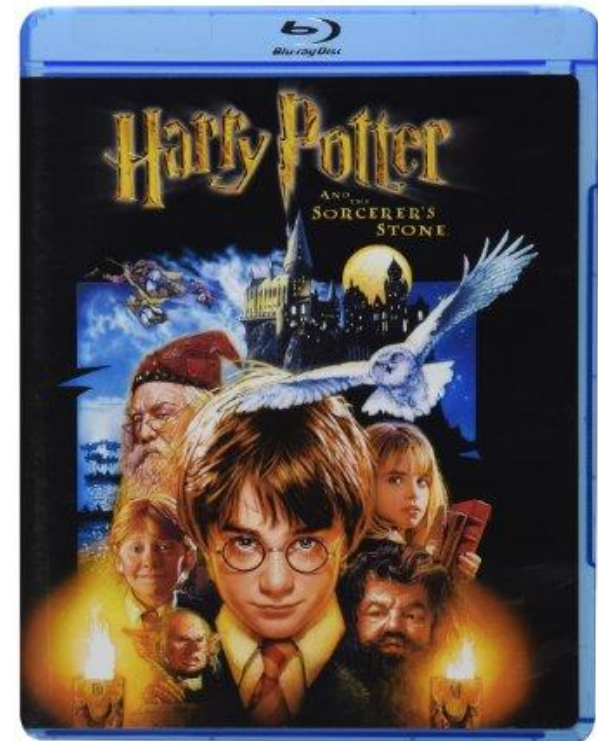
Analogy

Suppose you want to buy a Blu-ray movie from Amazon (~40GB)

Two options:

1. Download
2. 2-day Prime Shipping

Which is faster?



Buying the Entire Series

What if you want more movies?

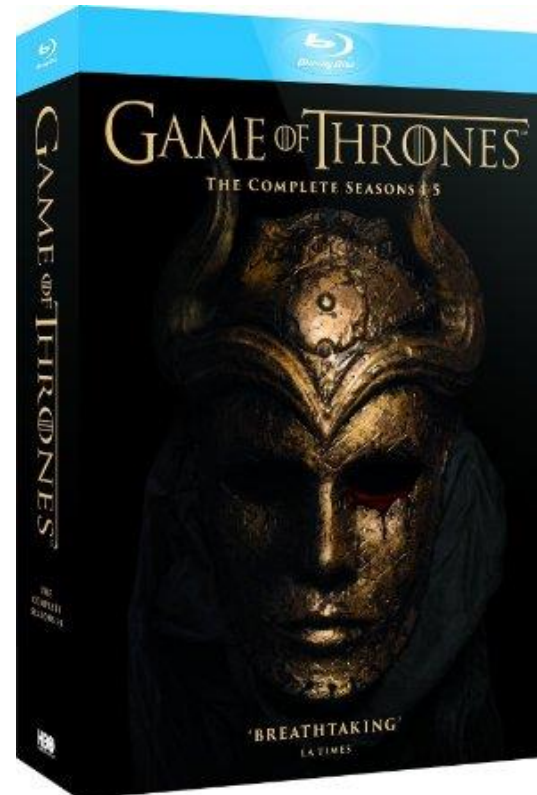
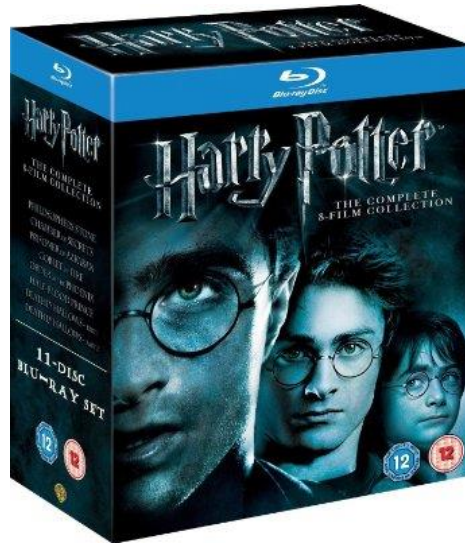
- 8 Blu-ray discs
- ~320 GB

Which is faster?

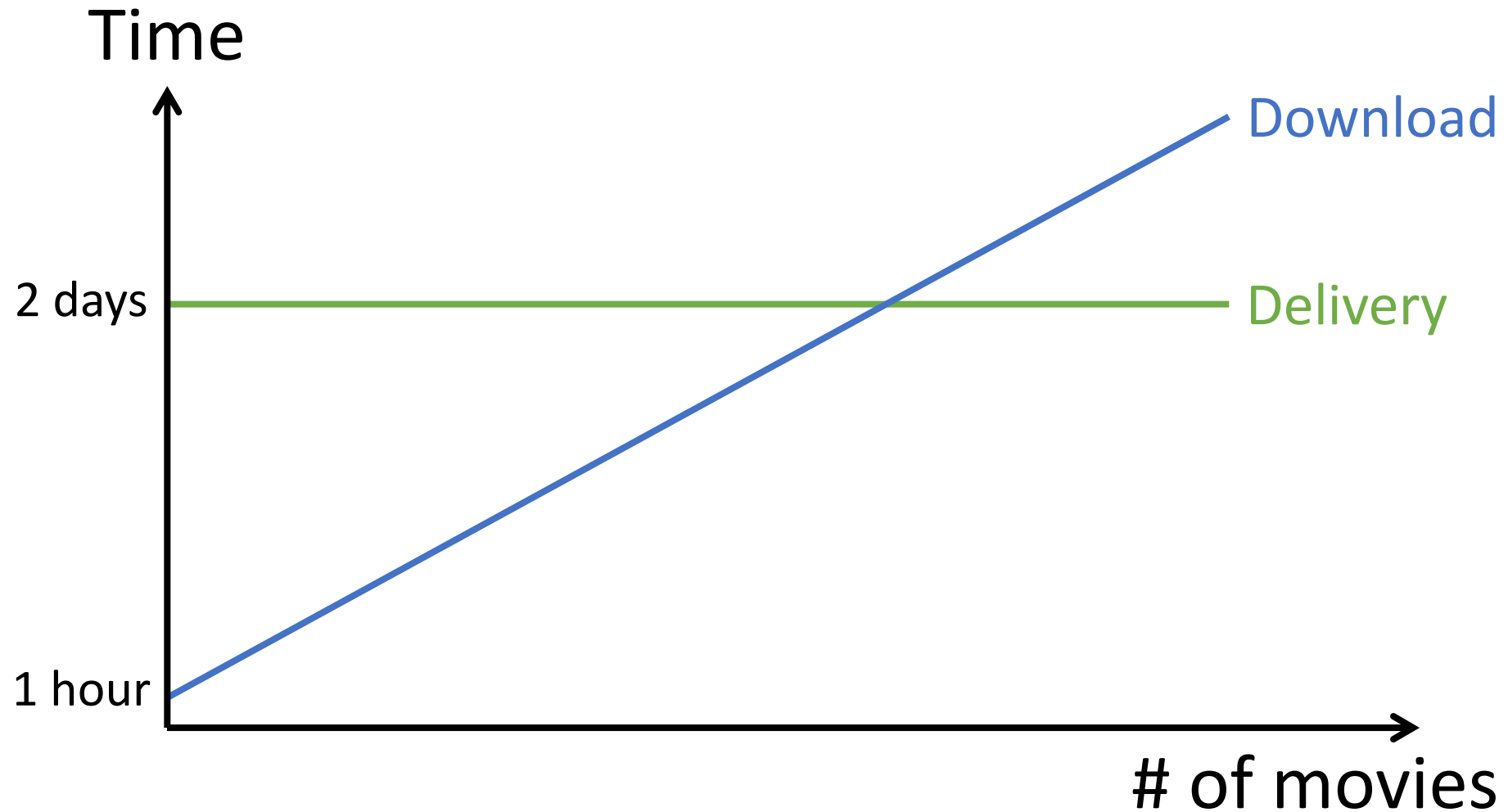
1. Download, or
2. 2-day delivery



Even more movies?



Download vs Delivery



We want to ask questions like:

`factorial(5) → factorial(10) ?`

`fib(10) → fib(20)?`

How much more time?

2x?

How much more space?

Same?

4x?

Order of Growth is **NOT** the **absolute**
time or space a program takes to run

Order of Growth is the proportion of growth of the time/space of a program w.r.t. the growth of the input

Formal Definition

Let n denote size of the problem.

Let $R(n)$ denote the resources needed.

Definition:

$R(n)$ has order of growth $\Theta(f(n))$ written

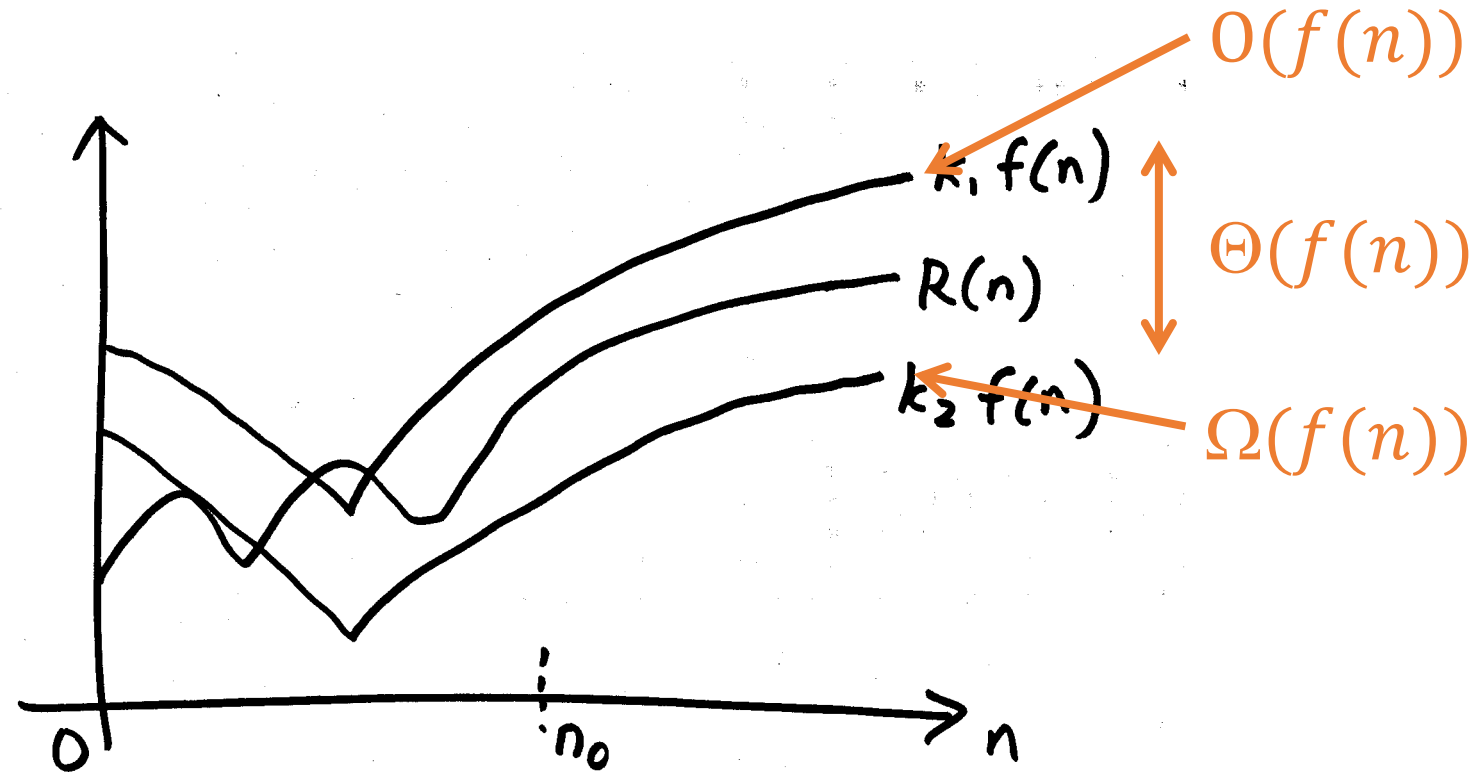
$$R(n) = \Theta(f(n))$$

If there are positive constants k_1 and k_2 such that

$$k_1 f(n) \leq R(n) \leq k_2 f(n)$$

for any sufficiently large value of n

Diagram



For $n \geq n_0$, $R(n)$ is sandwiched between

Some common $f(n)$

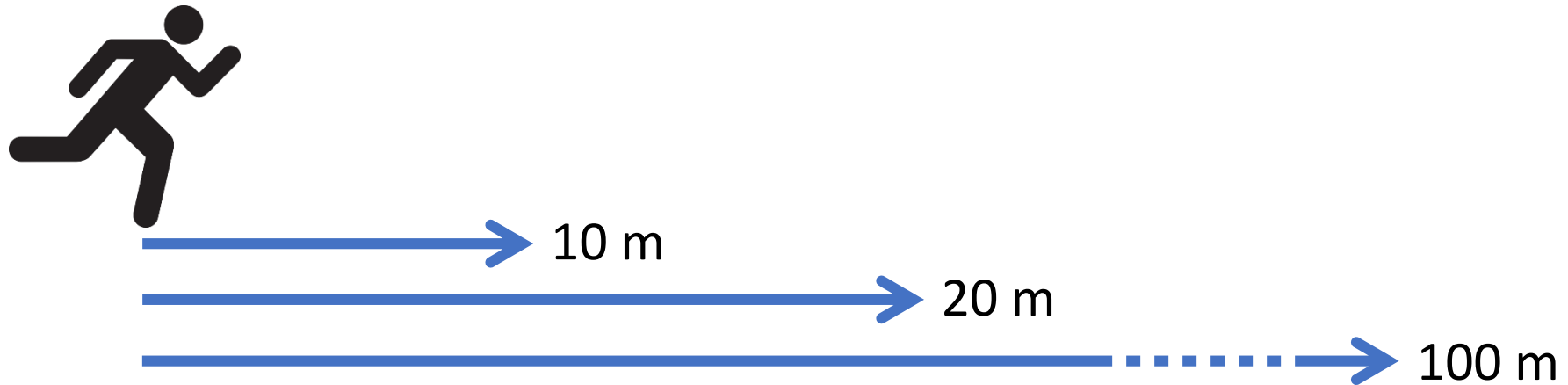
- 1
- n
- n^2
- n^3
- $\log n$
- $n \log n$
- 2^n

Intuitively

If n is doubled
(i.e. increased to $2n$)
then $R(n)$
(the resource required),
is increased to $f(2n)$

Another analogy

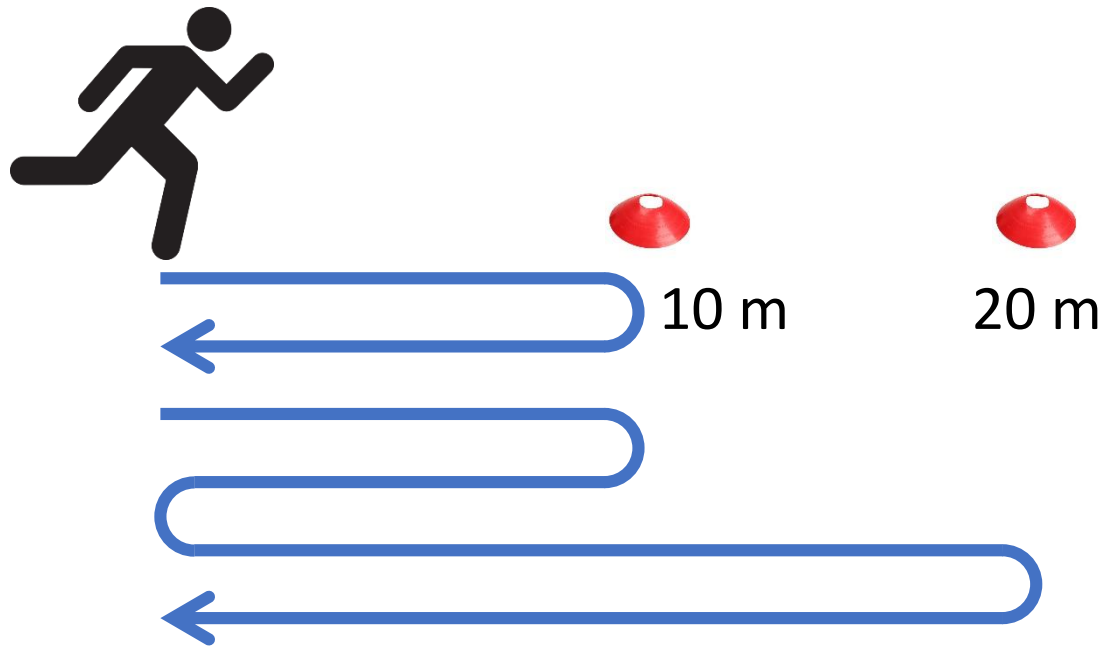
- Suppose you can run 10 m in 1.5 secs



- Time is linear to distance

Shuttle Run

- Run and return



- Time is _____ to distance

Recap: Recursive Factorial

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

Order of growth?

1. Time
2. Space

Recursive process

factorial(5)

5 * factorial(4)

5 * (4 * factorial(3))

5 * (4 * (3 * factorial(2)))

5 * (4 * (3 * (2 * factorial(1))))

5 * (4 * (3 * (2 * 1)))

5 * (4 * (3 * 2))

5 * (4 * 6)

5 * 24

120

- Time \propto #operations
- Linearly proportional to n

Recursive process

factorial(5)

5 * factorial(4)

5 * (4 * factorial(3))

5 * (4 * (3 * factorial(2)))

5 * (4 * (3 * (2 * factorial(1))))

5 * (4 * (3 * (2 * 1)))

5 * (4 * (3 * 2))

5 * (4 * 6)

5 * 24

120

- Space \propto #pending operations
- Linearly proportional to n

Recursive Factorial

factorial(5)

5 * factorial(4)

5 * (4 * factorial(3))

5 * (4 * (3 * factorial(2)))

5 * (4 * (3 * (2 * factorial(1))))

5 * (4 * (3 * (2 * 1)))

5 * (4 * (3 * 2))

5 * (4 * 6)

5 * 24

120

Time: $O(n)$ Linear

Space: $O(n)$ Linear

Iterative Factorial

```
def factorial(n):  
    product, counter = 1, 1  
    while counter <= n:  
        product = (product *  
                    counter)  
        counter = counter + 1  
    return product
```

```
factorial(6)
```

product	counter
1	1
1	2
2	3
6	4
24	5
120	6
720	7

Iterative process

product: 720

counter: 7

Time (# of steps):

- linearly proportional to n

Space (memory):

- constant
- no deferred operations
- All information contained in 2 variables (old values overwritten by new)

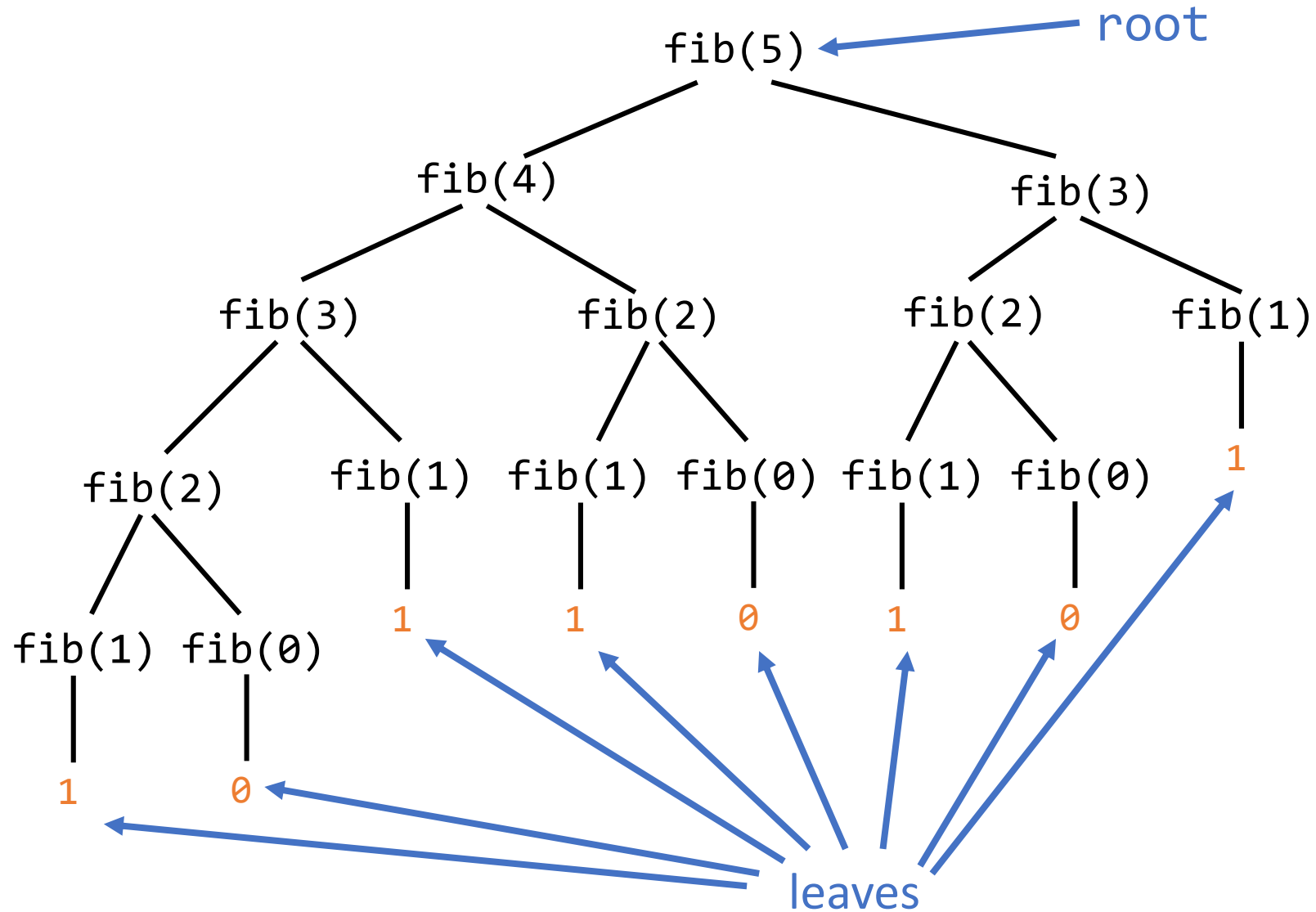
Time: $O(n)$ Linear
Space: $O(1)$ Constant

Recap: Fibonacci

$$fib(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ fib(n-1) + fib(n-2) & n > 1 \end{cases}$$

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


Tree recursion



Fibonacci

- Number of leaves in tree is $fib(n + 1)$
- Can be shown that $fib(n)$ is the closest integer to $\frac{\Phi^n}{\sqrt{5}}$
 - Where $\Phi = \frac{1+\sqrt{5}}{2} \approx 1.6180$
 - called the golden ratio
- Therefore time taken is $\approx \Phi^n$
 - (exponential in n)

Tree recursion

- Time:
 - Proportional to number of leaves, i.e.,
exponential in n .
- Space (memory):
 - Proportional to the depth of the tree, i.e.,
linear in n .

General form

Suppose a computation \mathcal{C} takes $3n + 5$ steps to complete, what is the order of growth?

$$O(3n + 5) = O(n)$$

Take the largest term.
Drop the constants.

Another Example

How about $3^n + 4n^2 + 4$?

Order of growth

$$\begin{aligned} &= O(3^n + \underbrace{4n^2 + 4}_{\text{drop}}) \\ &= O(3^n) \end{aligned}$$

Tips

- Identify dominant terms, ignore smaller terms
- Ignore additive or multiplicative constants
 - $4n^2 - 1000n + 300000 = O(n^2)$
 - $\frac{n}{7} + 200n \log n = O(n \log n)$
- Note: $\log_a b = \frac{\log_c b}{\log_c a}$
 - So base is not important

More tricks in CS1231, CS3230

Some involve sophisticated proofs

For now...

Count the number of “basic computational steps”.

- Identify the basic computation steps
- Try a few small values of n
- Extrapolate for really large n
- Look for “worst case” scenario

Numeric example

n	$\log n$	$n \log n$	n^2	n^3	2^n
1	0	0	1	1	2
2	0.69	1.38	4	8	4
3	1.098	3.29	9	27	8
10	2.3	23.0	100	1000	1024
20	2.99	59.9	400	8000	10^6
30	3.4	109	900	27000	10^9
100	4.6	461	10000	10^6	1.2×10^{30}
200	5.29	1060	40000	8×10^6	1.6×10^{60}
300	5.7	1710	90000	27×10^6	2.03×10^{90}
1000	6.9	6910	10^6	10^9	1.07×10^{301}
2000	7.6	15200	4×10^6	8×10^9	?
3000	8	24019	9×10^6	27×10^9	?
10^6	13.8	13.8×10^6	10^{12}	10^{18}	?

13.7 billion years $\approx 2^{59}$ seconds

Time: how long it takes to run a program

Space: how much memory do we need to run the program

pythontutor.com



Moral of the story

Different ways of performing a computation (algorithms) can consume **dramatically** different amounts of resources.

Recursion Revisited

- Solve the problem for a simple (base) case
- Express (divide) a problem into one or more smaller similar problems
- Similar to

Mathematical Induction

Comparison

Mathematical Induction

- Start with a base case b
- Assume k works, derive a function to show $k + 1$ also works
- Therefore, it must be true for all cases $\geq b$

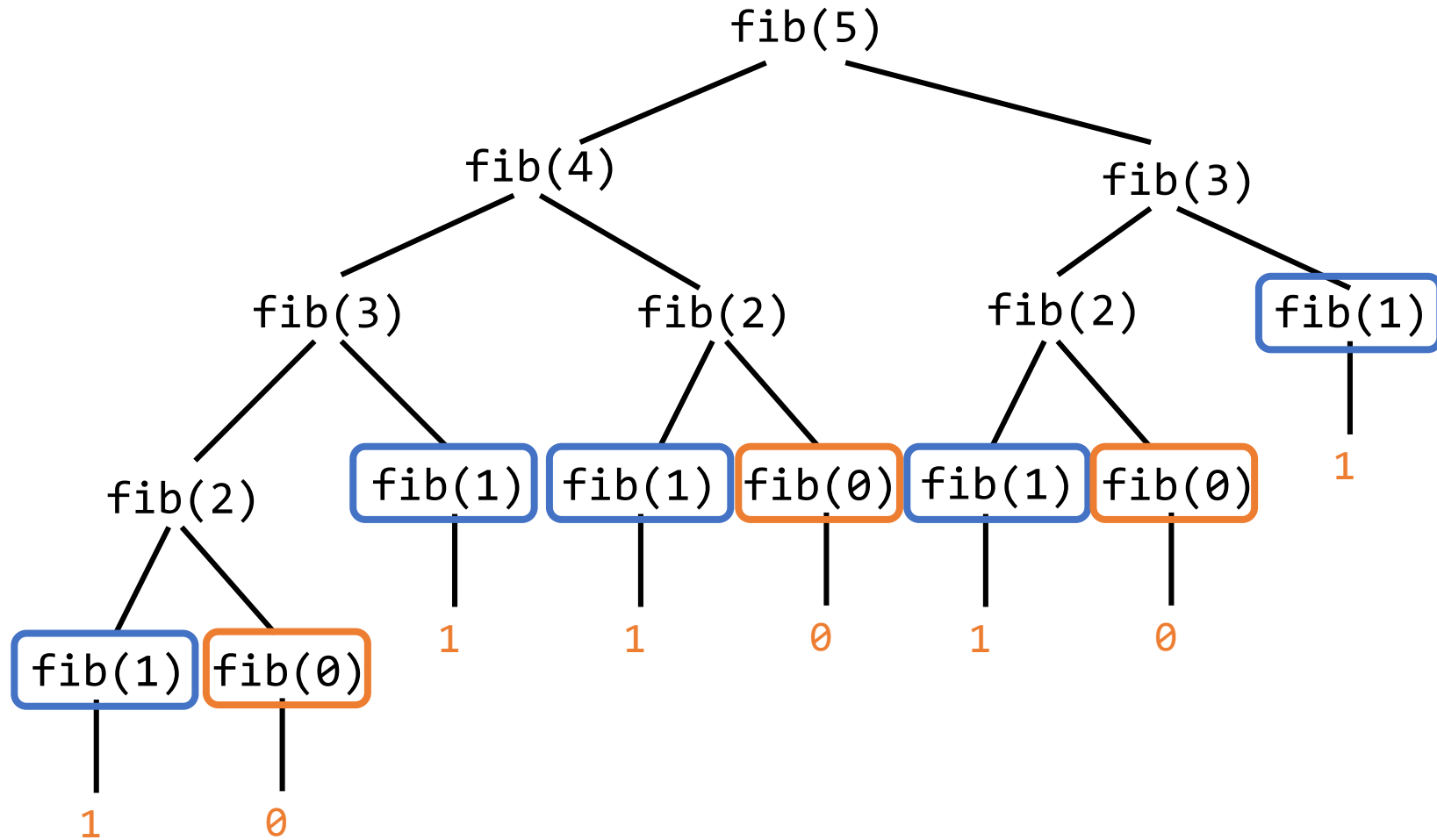
Recursion

- Find base case(s) b where we can just state the answer
- Derive a function to express the problem of size n as sub-problems of $k < n$
- The function can therefore solve all $n \geq b$

Sometimes it may be possible that you will need more than one base case?

When? Why?

Tree recursion



Other times you may have to express a problem in another form and the other form back in the present form
(mutual recursion)

- E.g. `sin` and `cos`

More Examples

Greatest Common Divisor

HCF

The *GCD* of two numbers a and b , is the largest positive integer that divides both a and b without remainder.

Greatest Common Divisor

Naïve Algorithm:

Given two numbers a and b

Start with 1.

Check if it divides both a and b .

Try 2, then 3, and so on... until you reach a or b .

Greatest Common Divisor

Euclid's Algorithm:

Given two numbers a and b , where $a = b \cdot Q + r$ (r the remainder of the division), then we have

$$GCD(a, b) = GCD(b, r), \forall a, b > 0$$

$$GCD(a, 0) = a$$

Greatest Common Divisor

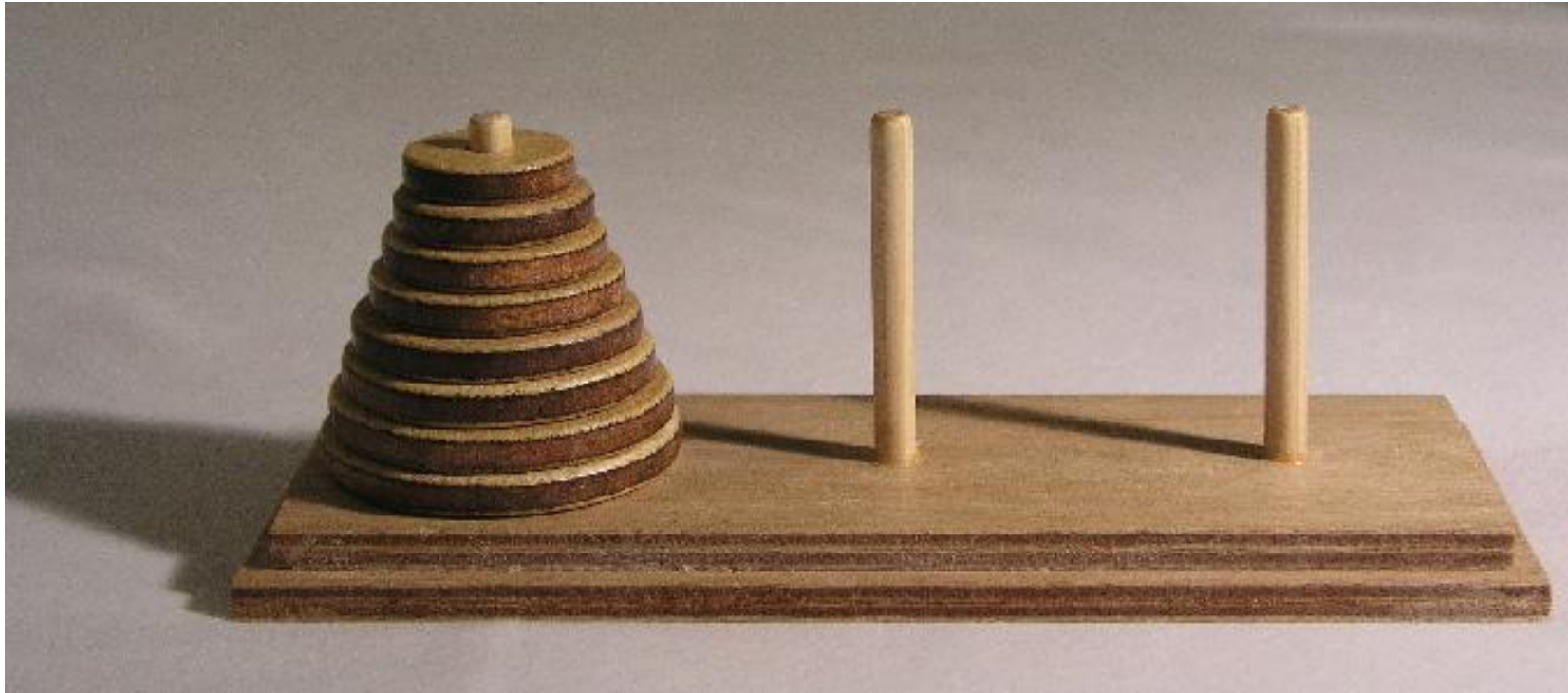
```
def gcd(a, b):  
    if (b == 0):  
        return a  
    else:  
        return gcd(b, a % b)
```

$$GCD(a, b) = GCD(b, r), \forall a, b > 0$$

$$GCD(a, 0) = a$$

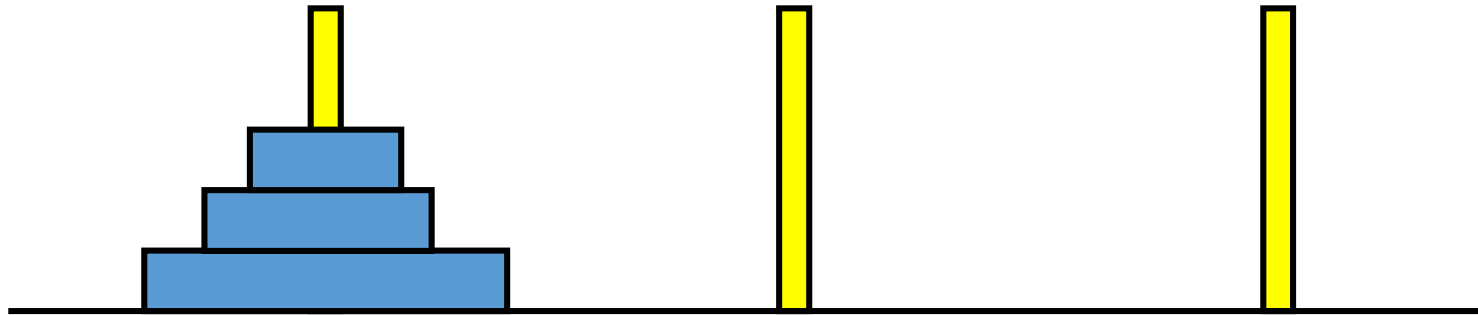
```
GCD(206, 40) = GCD(40, 6)  
              = GCD(6, 4)  
              = GCD(4, 2)  
              = GCD(2, 0)  
              = 2
```


Tower of Hanoi



Towers of Hanoi

Goal: Move all discs from one stick to another

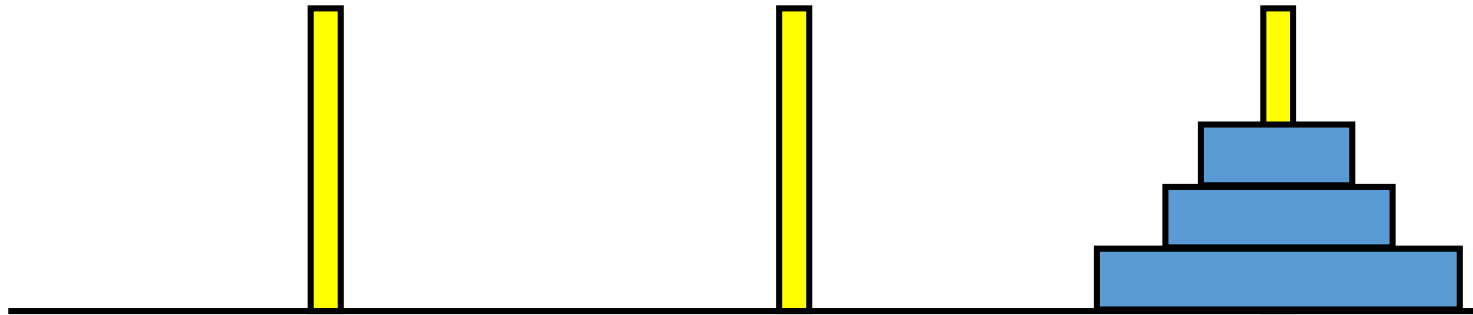


Rules:

1. Can only move one disc at a time
2. Cannot put a larger disc over a smaller disc

Towers of Hanoi

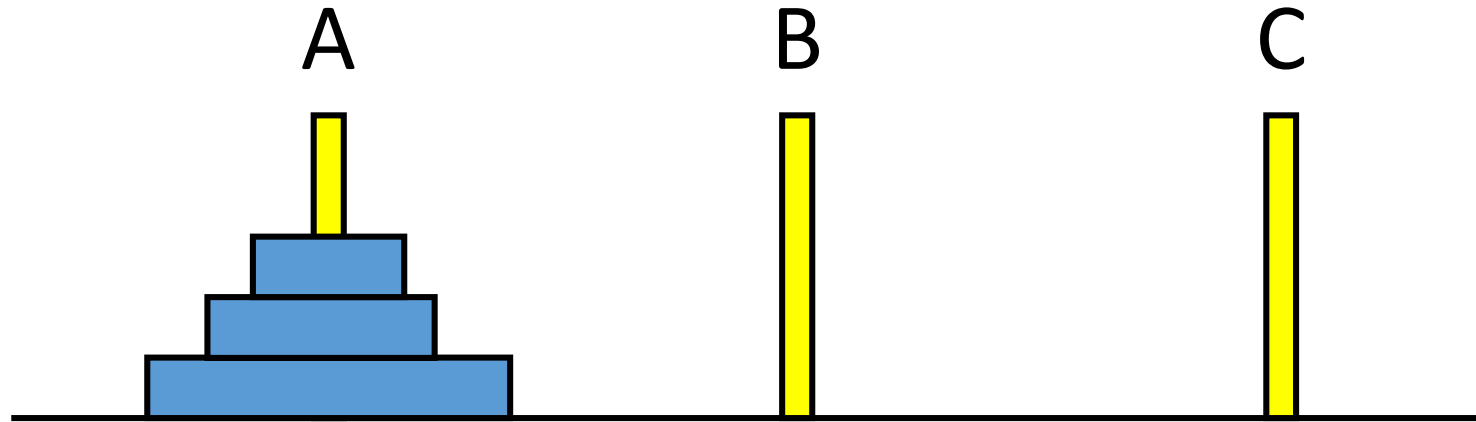
Goal: Move all discs from one stick to another



Rules:

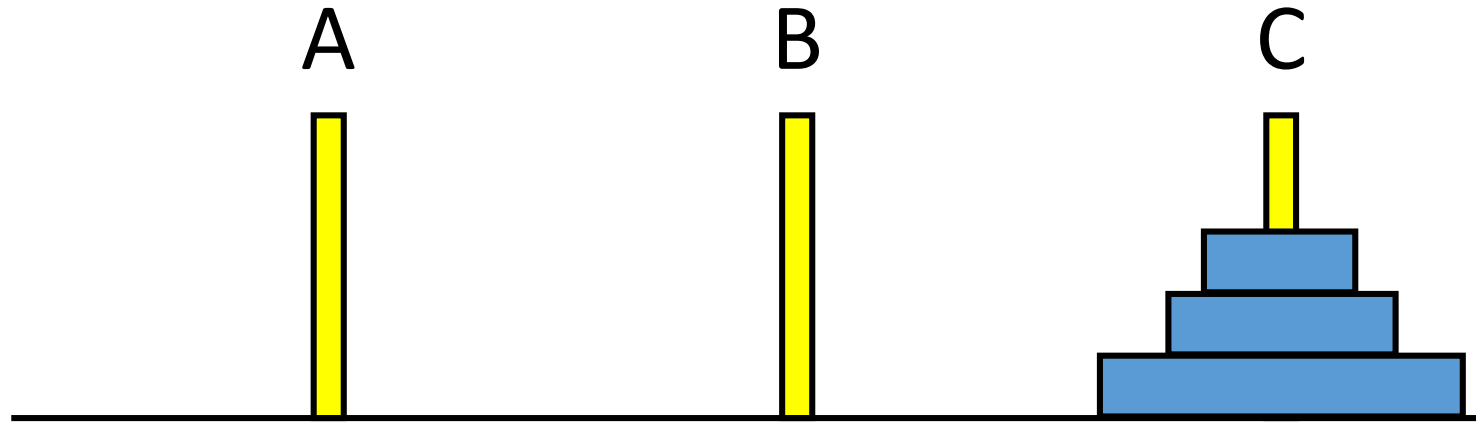
1. Can only move one disc at a time
2. Cannot put a larger disc over a smaller disc

Towers of Hanoi



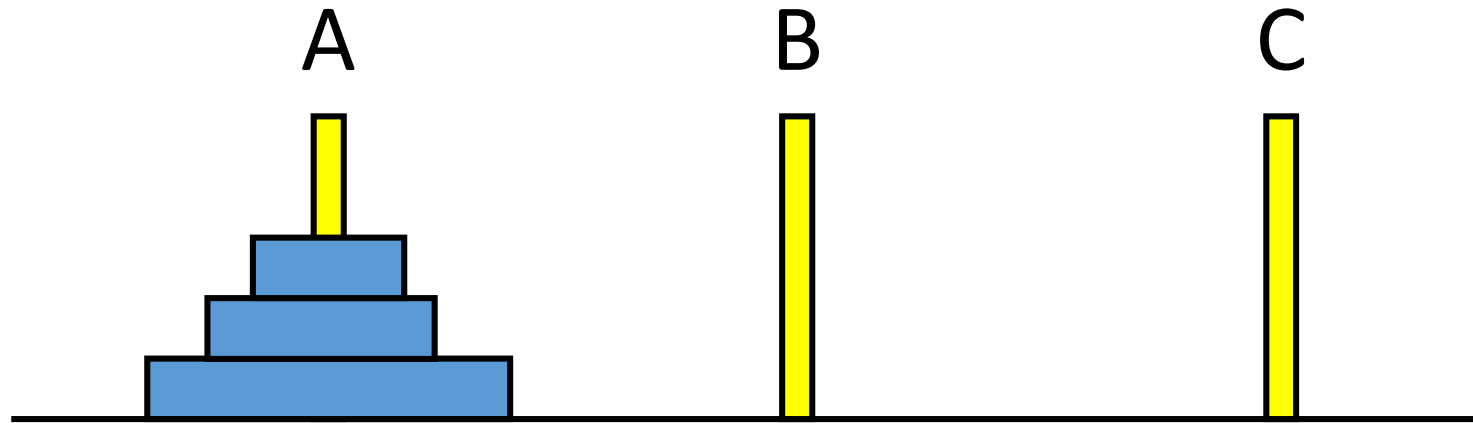
Suppose we know how to move 3
discs from A to C

Towers of Hanoi



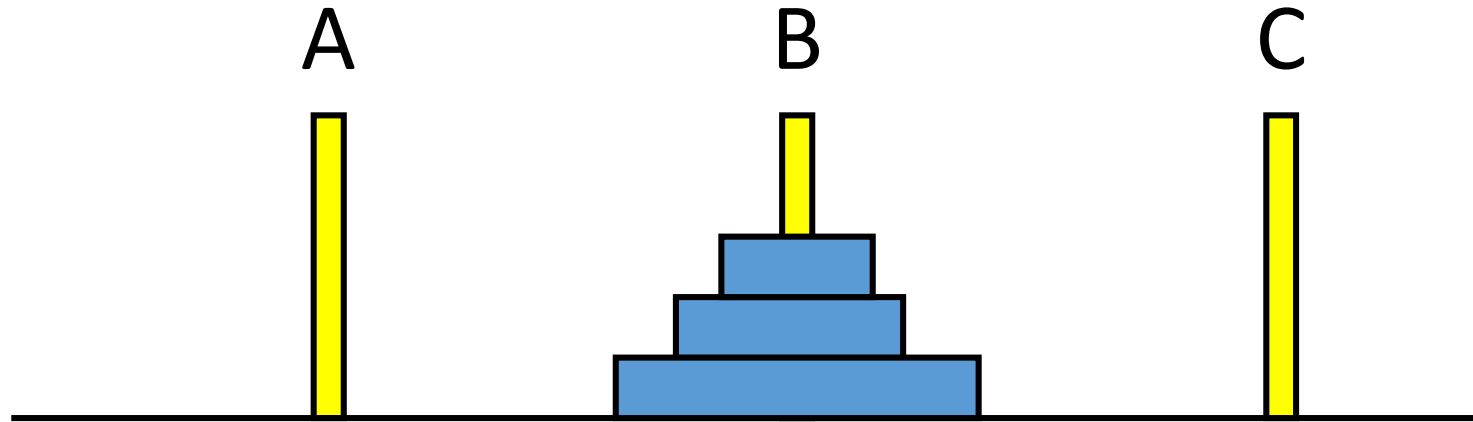
Suppose we know how to move 3
discs from A to C

Towers of Hanoi



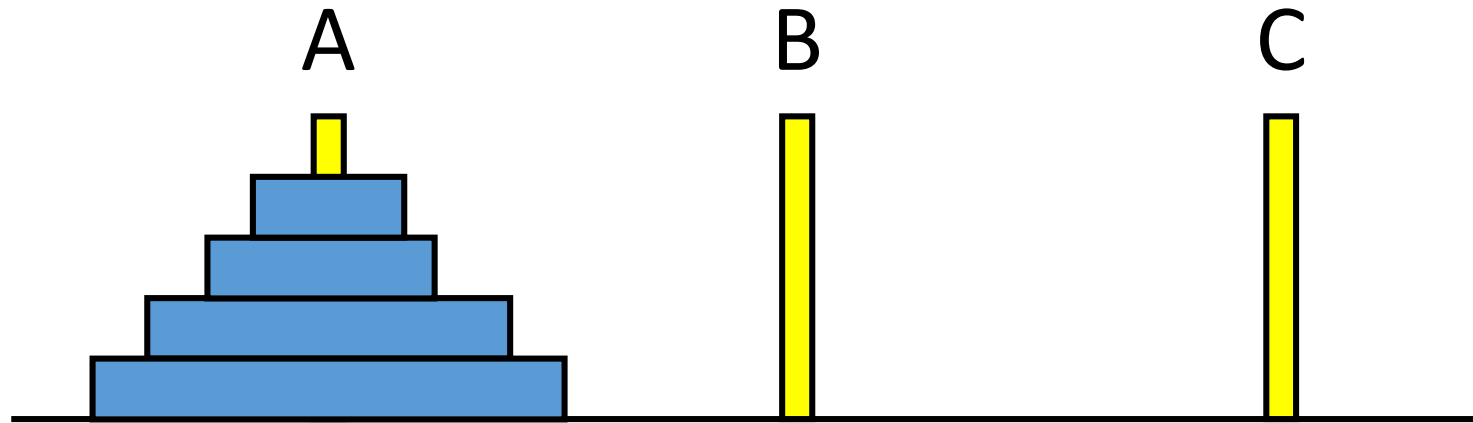
Claim: we can move 3 discs from
A to B. Why?

Towers of Hanoi



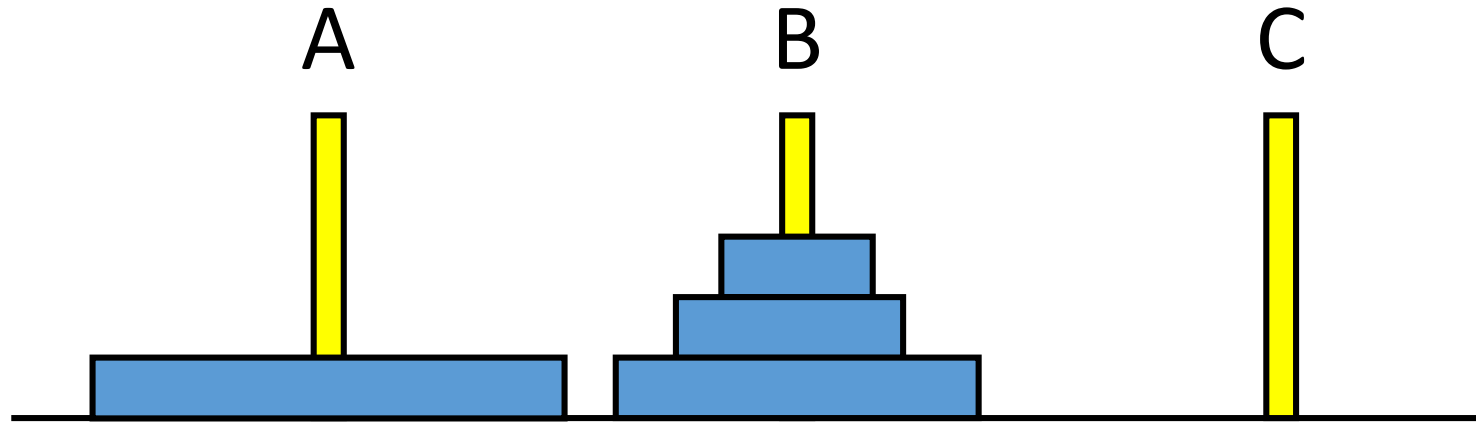
Claim: we can move 3 discs from
A to B. Why?

Towers of Hanoi



How to move 4 discs from A to C?

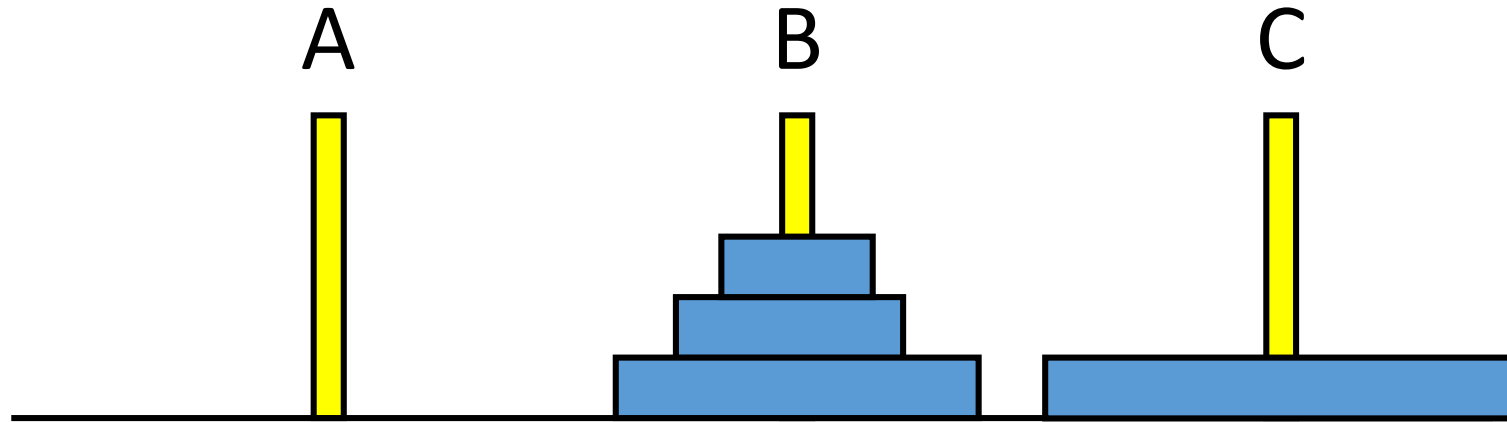
Towers of Hanoi



How to move 4 discs from A to C?

- Move 3 disc from A to B

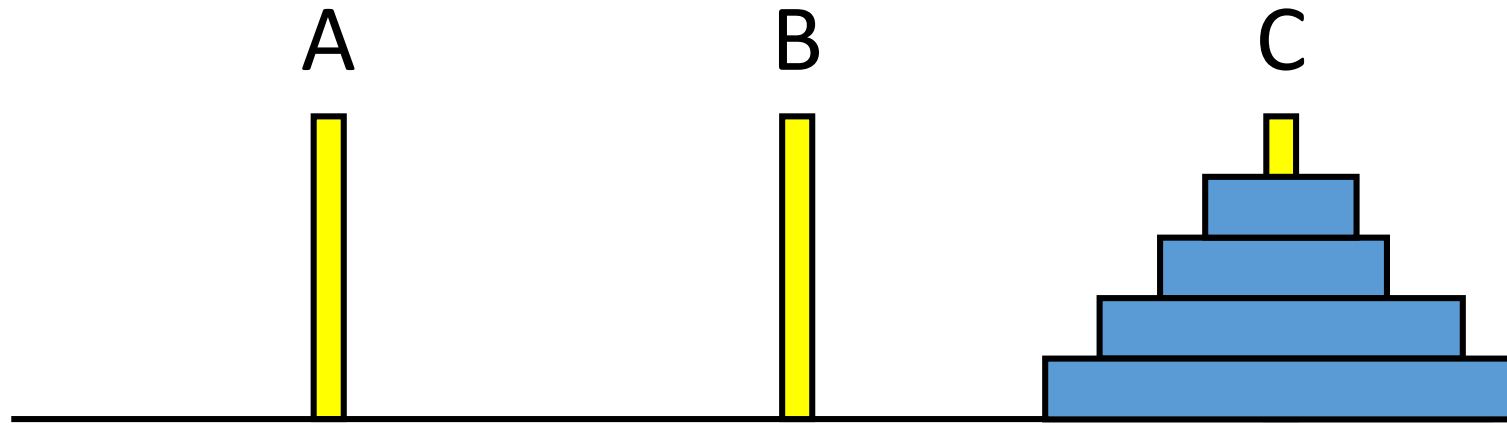
Towers of Hanoi



How to move 4 discs from A to C?

- Move 3 disc from A to B
- Move 1 disc from A to C

Towers of Hanoi



How to move 4 discs from A to C?

- Move 3 disc from A to B
- Move 1 disc from A to C
- Move 3 disc from B to C

Divided into smaller problem

- Move 4 discs → Move 3 discs
- Move 5 discs? → Move 4 discs
- Move n discs? → Move $n - 1$ discs

Recursion

1. Expressed (divided) the problem into one or more smaller problems

$$n = f(n - 1)$$

2. Solve the simple (base) case

- 1 disc?

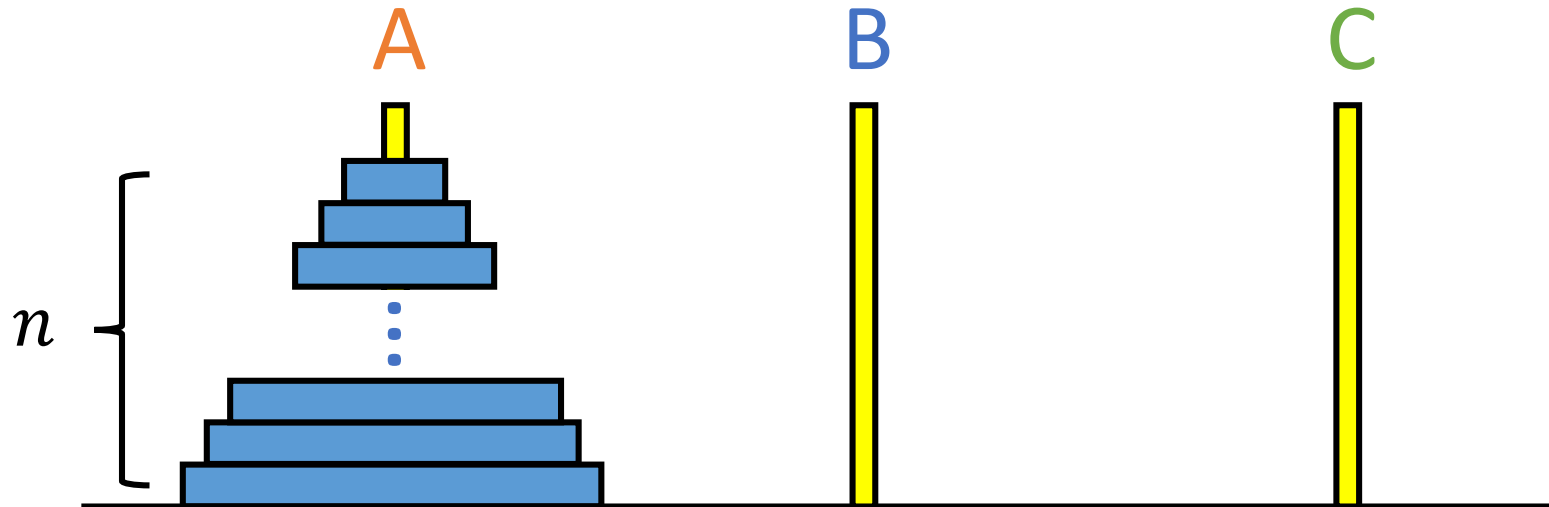
- 0 disc?

Move directly from X to Y

Do nothing

High Level Idea

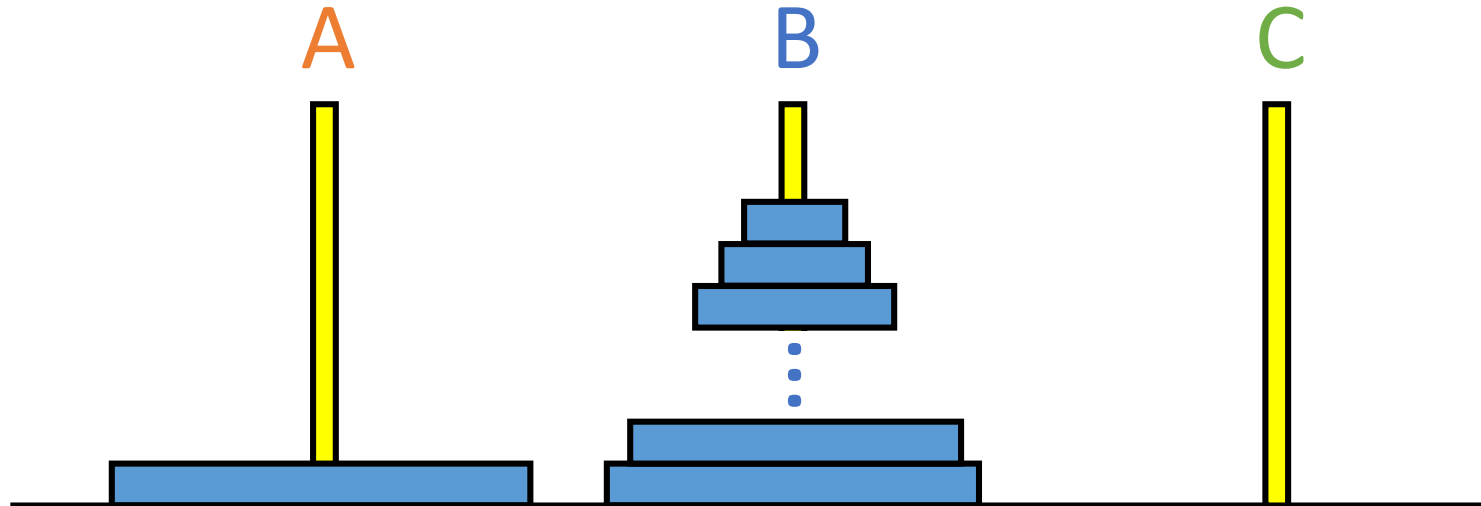
To move n discs from **A** to **C** using **B**



High Level Idea

To move n discs from **A** to **C** using **B**

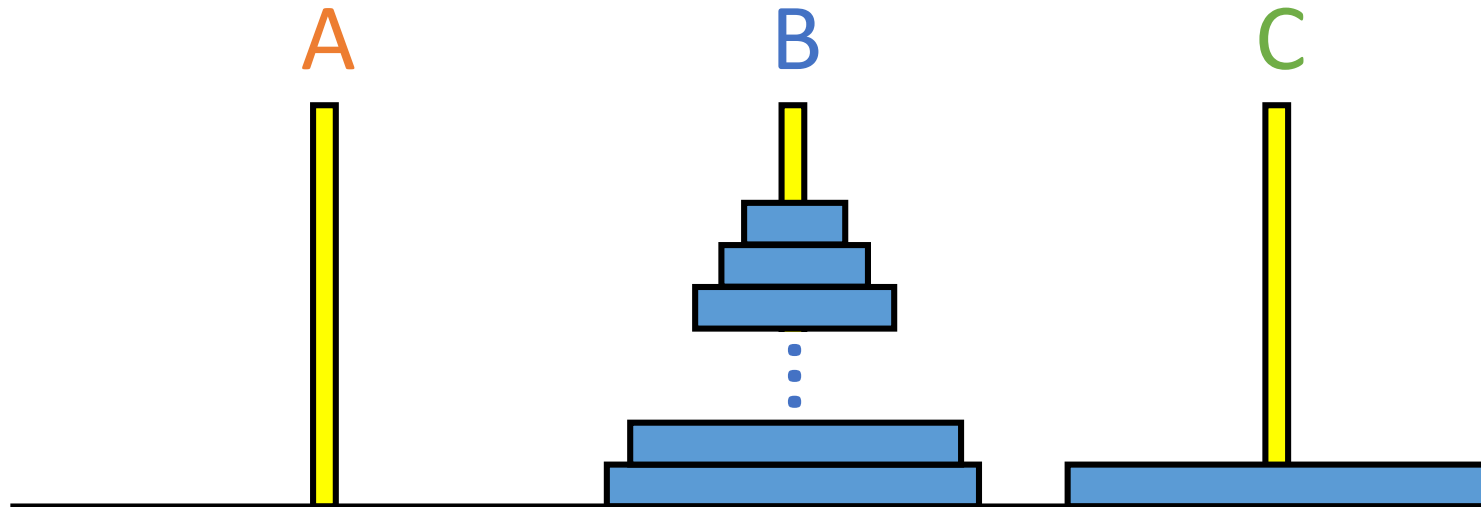
1. move $n - 1$ discs from **A** to **B** using **C**



High Level Idea

To move n discs from **A** to **C** using **B**

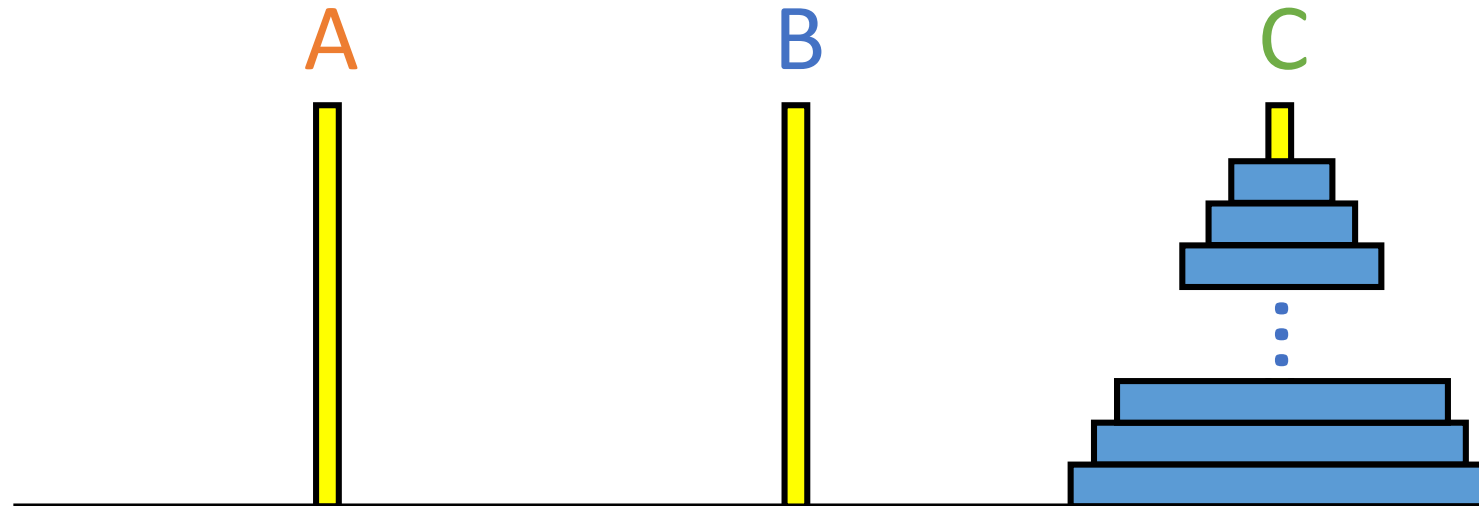
1. move $n - 1$ discs from **A** to **B** using **C**
2. move disc from **A** to **C**



High Level Idea

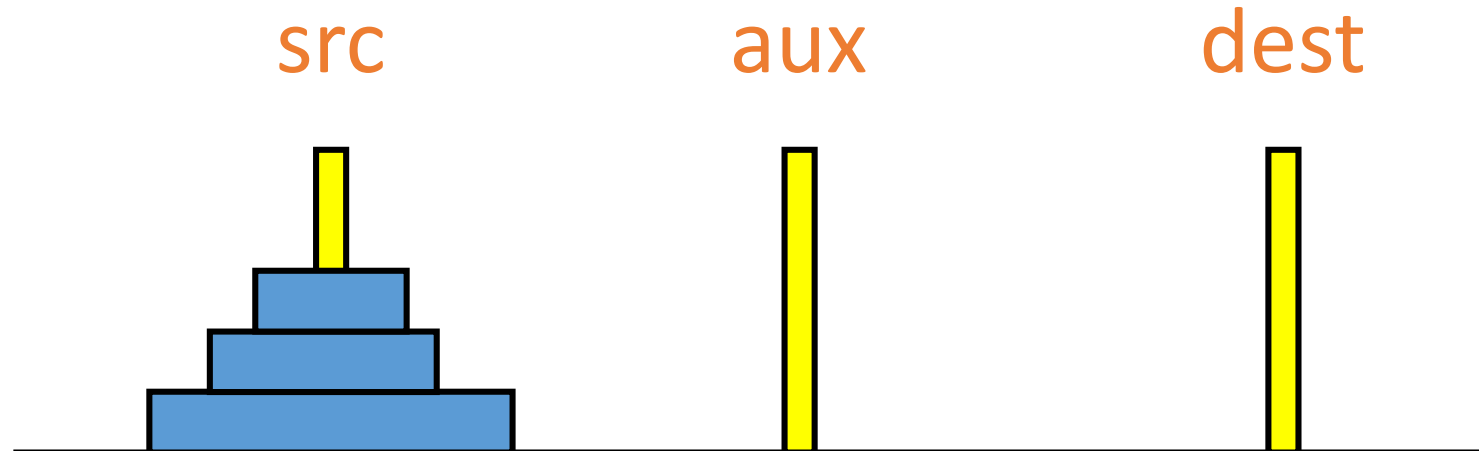
To move n discs from **A** to **C** using **B**

1. move $n - 1$ discs from **A** to **B** using **C**
2. move disc from **A** to **C**
3. move $n - 1$ discs from **B** to **C** using **A**



Towers of Hanoi

```
def move_tower(size, src, dest, aux):  
    if size == 1:  
        print_move(src, dest) # display the move  
    else:  
        move_tower(size-1, src, aux, dest)  
        print_move(src, dest)  
        move_tower(size-1, aux, dest, src)
```



Tower of Hanoi

```
def print_move(src, dest):  
    print("move top disk from ", src" to ", dest)
```

Another example

What does this function compute?

```
def foo(x, y):  
    if (y == 0):  
        return 1  
    else:  
        return x * foo(x, y-1)
```

This?

```
def power(b, e):  
    if (e == 0):  
        return 1  
    else:  
        return b * power(b, e-1)
```

Exponentiation (b^e)

```
def power(b, e):  
    if (e == 0):  
        return 1  
    else:  
        return b * power(b, e-1)
```

- Time requirement? $O(n)$
- Space requirement? $O(n)$

Can we do better?

Another way to express b^e

$$b^e = \begin{cases} 1, & e = 0 \\ (b^2)^{\frac{e}{2}}, & e \text{ is even} \\ b^{e-1} \cdot b, & e \text{ is odd} \end{cases}$$

Fast Exponentiation

```
def fast_expt(b, e):  
    if e == 0:  
        return 1  
    elif e % 2 == 0:  
        return fast_expt(b*b, e/2)  
    else:  
        return b * fast_expt(b, e-1)
```

$$b^e = \begin{cases} 1, & e = 0 \\ (b^2)^{\frac{e}{2}}, & e \text{ is even} \\ b^{e-1} \cdot b, & e \text{ is odd} \end{cases}$$

- Time requirement? $O(\log n)$
- Space requirement? $O(\log n)$

Can we do this iteratively?

Summary

- Recursion
 - Solve the problem for a simple (base) case
 - Express (divide) a problem into one or more smaller similar problems
- Iteration: `while` and `for` loops

Summary

- Order of growth:
 - Time and space requirements for computations
 - Different ways of performing a computation (algorithms) can consume dramatically different amounts of resources.
 - Pay attention to efficiency!

Something to think about....

- Can you write a recursive function `sum_of_digits` that will return the sum of digits of an arbitrary positive integer?
- How about a recursive function `product_of_digits` that will return the product of the digits?

Notice a pattern?

How would you write a function that computed the sum of square roots of the digits of a number?

Why is Python Cool?

Ask your friends in CS1010 how they
would solve these problems in C.

