

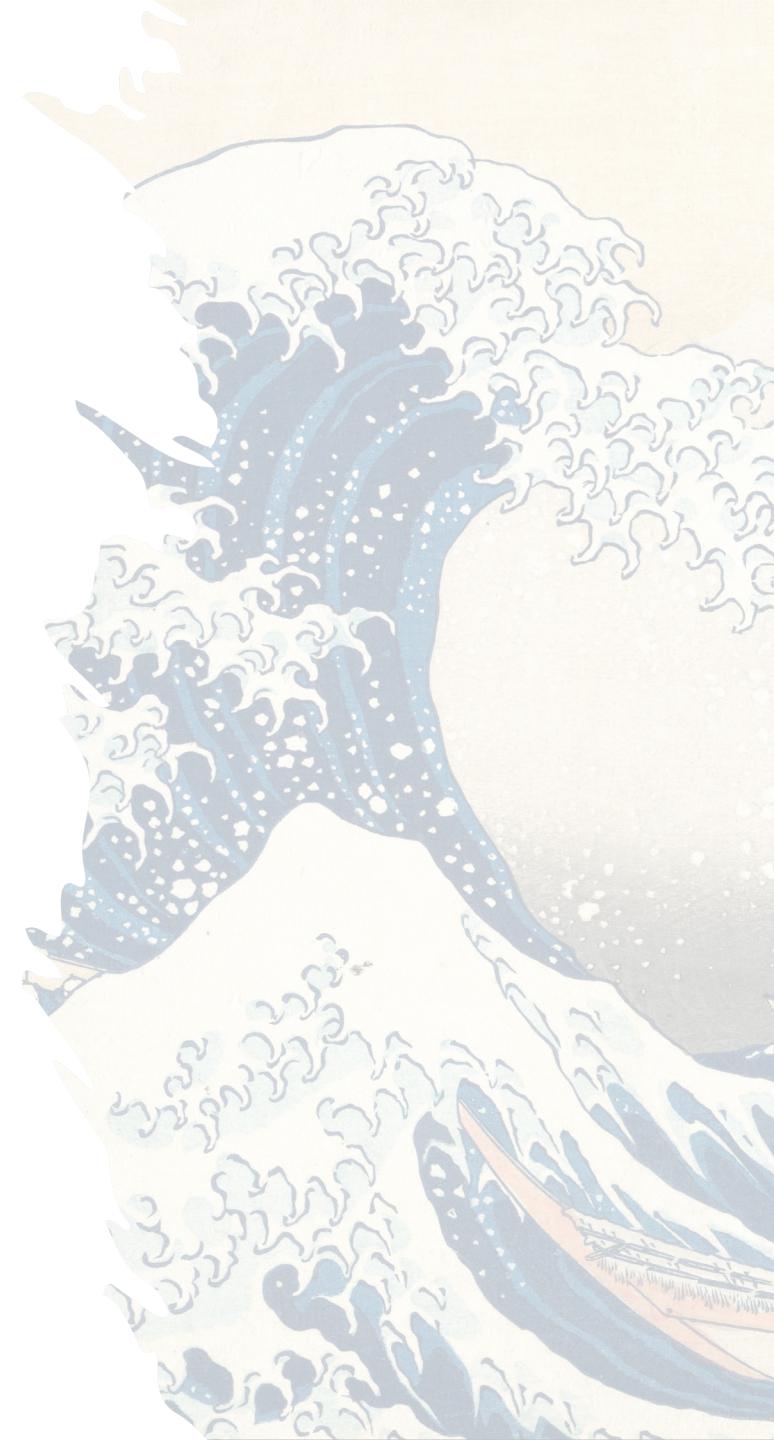


Stelios Sotiriadis

3. Multiprocessing with Python

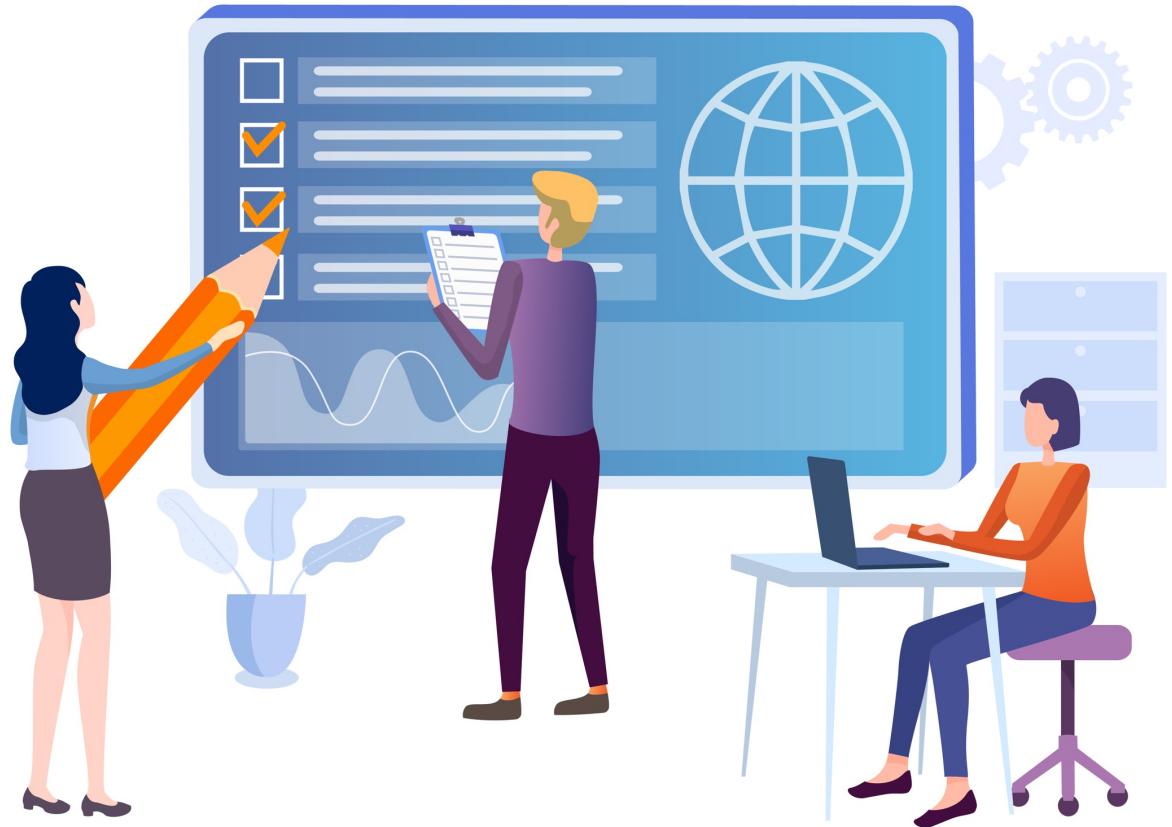
Agenda

- ▶ Finalise complexity: Dynamic programming
- ▶ Introduction to multiprocessing with Python
 - Serial vs Parallel processing
 - When to use?
 - Processes and Threads
- ▶ Lab 3 – using the `multiprocessing` library



Quiz of the day

Get ready!



Complexity!

Algorithm	Time	Space
Linear search	$O(n)$	$O(1)$
Binary search	$O(\log n)$	$O(1)$
Bubble sort	$O(n^2)$	$O(1)$
Insertion sort	$O(n^2)$	$O(1)$
Merge sort	$O(n \log n)$	$O(n)$
Tim sort	$O(n \log n)$	$O(n)$
Knapsack fractional	$O(n \log n)$	$O(n)$

We can still do
better!



Dynamic programming





The Romanesco flower!

- ▶ Romanesco has flowers that form perfect geometric spiral patterns that repeat.
- ▶ Each spiral bud is composed of a series of smaller buds
 - All buds are arranged in yet another spiral that continues at smaller levels
- ▶ The number of spirals on Romanesco's head follows the **Fibonacci sequence!**

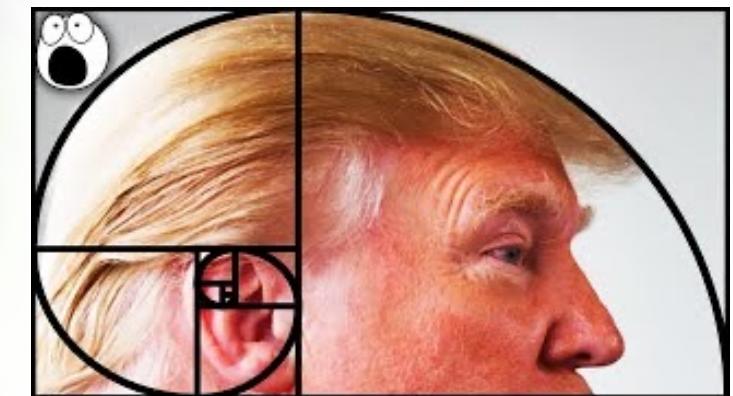


Other examples

Fibonacci symmetry
algorithm underlies our
perception of attractiveness



Reflects the
growth processes
in nature



What is the Fibonacci sequence?

- ▶ The Fibonacci Sequence is a series of numbers:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

- ▶ The sequence starts with **zero** and **one** and is a steadily increasing series where each number equals the sum of the preceding two numbers.
- ▶ Example: Fibonacci F_3 is found by adding $F_2 + F_1$

$$F_3 = F_2 + F_1 = 1+1 = 2$$

$$F_2 = F_1 + F_0 = 1+0 = 1$$

$$F_1 = 1 \quad F_0 = 0$$

Quiz!

- ▶ What is the Fibonacci of 7?
 - Can you do it without pen and paper? ☺

$$F_3 = F_2 + F_1 = 1+1 = 2$$

$$F_2 = F_1 + F_0 = 1+0 = 1$$

$$F_1 = 1 \quad F_0 = 0$$

$$F_0 = 0$$

$$F_1 = 1$$

$$F_2 = 1$$

$$F_3 = 2$$

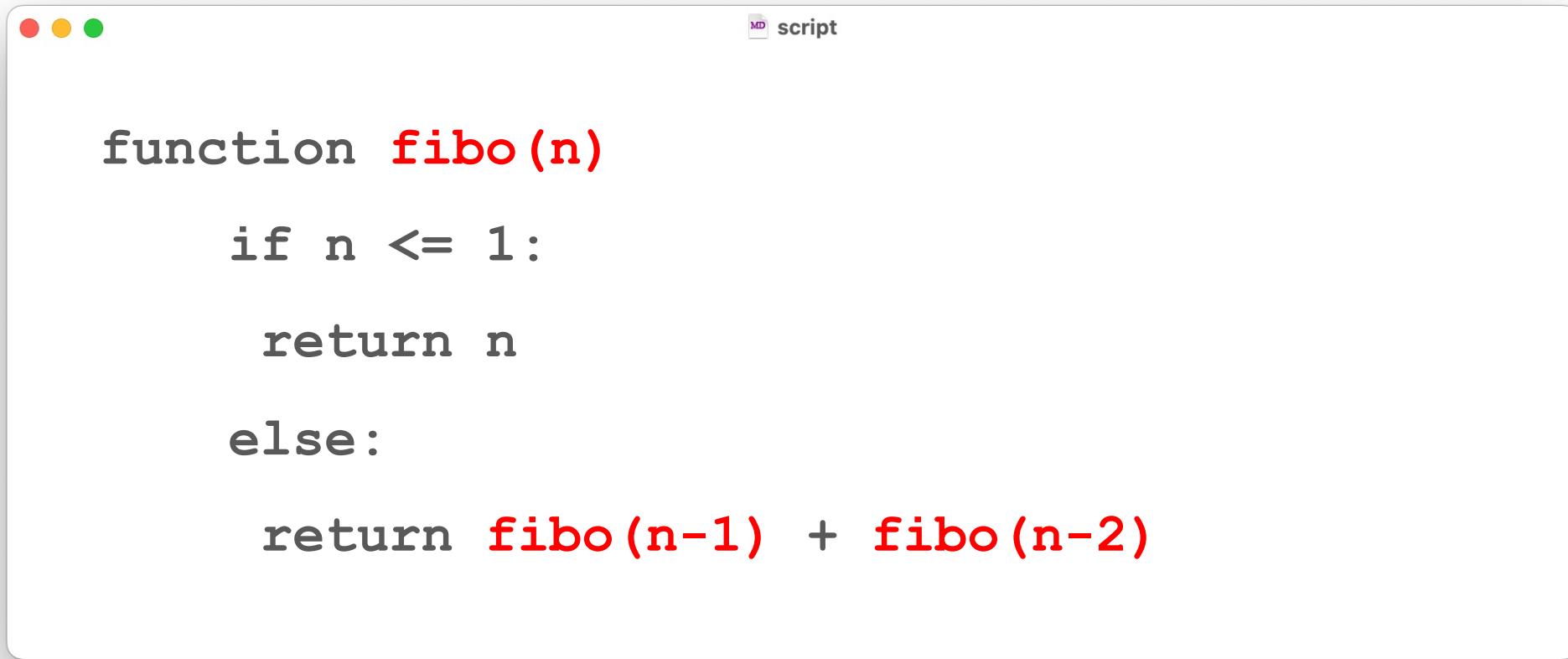
$$F_4 = 3$$

$$F_5 = 5$$

$$F_6 = 8$$

$$\mathbf{F_7 = 13}$$

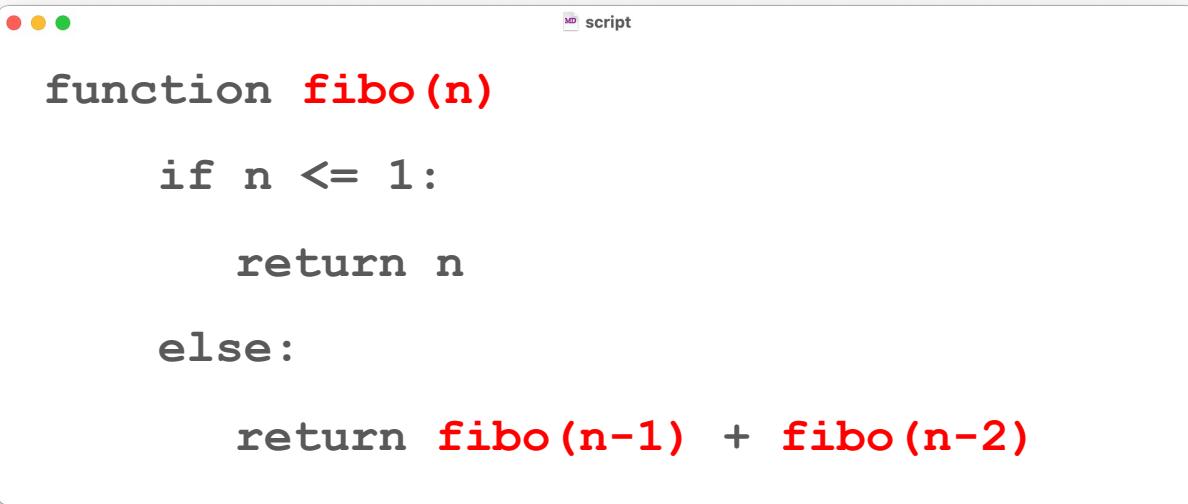
How to calculate the Fibonacci sequence?



A screenshot of a code editor window titled "script". The window has three colored window control buttons (red, yellow, green) at the top left. The code inside the editor is as follows:

```
function fibo(n)
    if n <= 1:
        return n
    else:
        return fibo(n-1) + fibo(n-2)
```

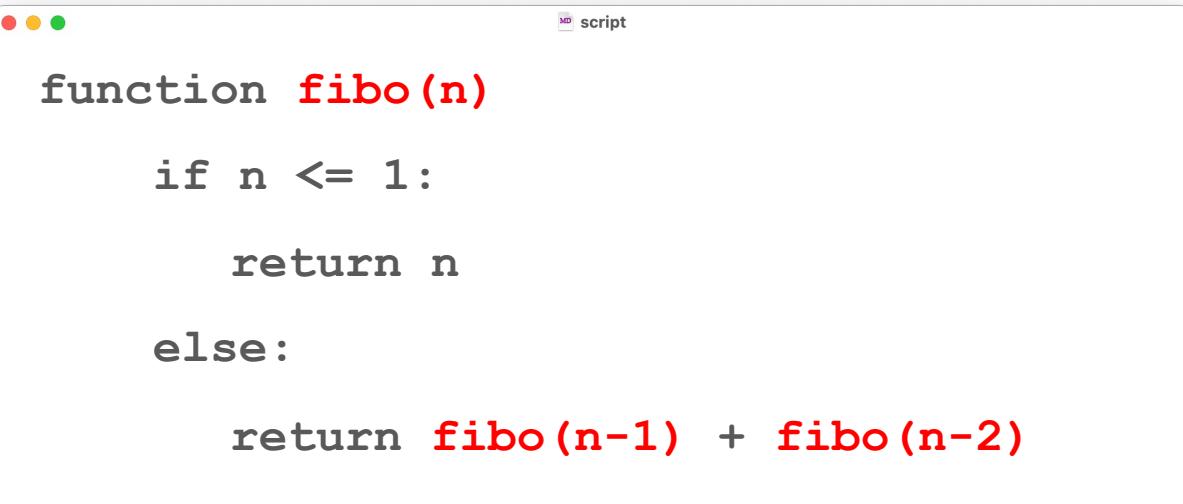
How to calculate the Fibonacci sequence?



```
function fibo(n)
    if n <= 1:
        return n
    else:
        return fibo(n-1) + fibo(n-2)
```

- ▶ If **n** = 0, then what is the **fibo(0)**?
 - Answer: 0
- ▶ If **n** = 1, then what is the **fibo(1)**?
 - Answer: 1

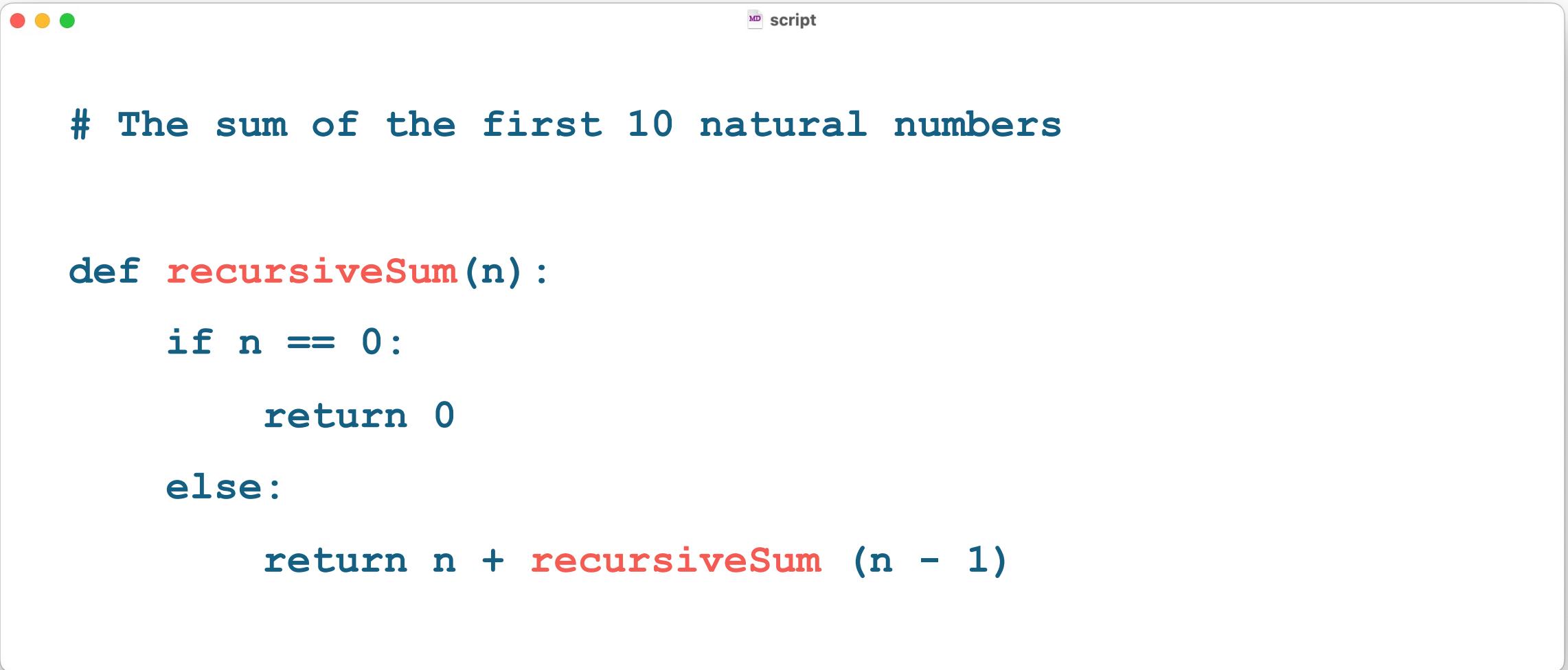
How to calculate the Fibonacci sequence?



```
function fibo(n)
    if n <= 1:
        return n
    else:
        return fibo(n-1) + fibo(n-2)
```

- ▶ If $n = 2$, then what is the **fibo(2)**?
 - Answer: $\text{fibo}(1) + \text{fibo}(0)$
 $= 1 + 0 = 1$
- ▶ If $n = 3$, then what is the **fibo(3)**?
 - Answer: $\text{fibo}(2) + \text{fibo}(1)$
 $= 1 + 1 = 2$

What is recursion in programming?



A screenshot of a code editor window titled "script". The window has three colored window control buttons (red, yellow, green) at the top-left. The code inside the editor is as follows:

```
# The sum of the first 10 natural numbers

def recursiveSum(n):

    if n == 0:

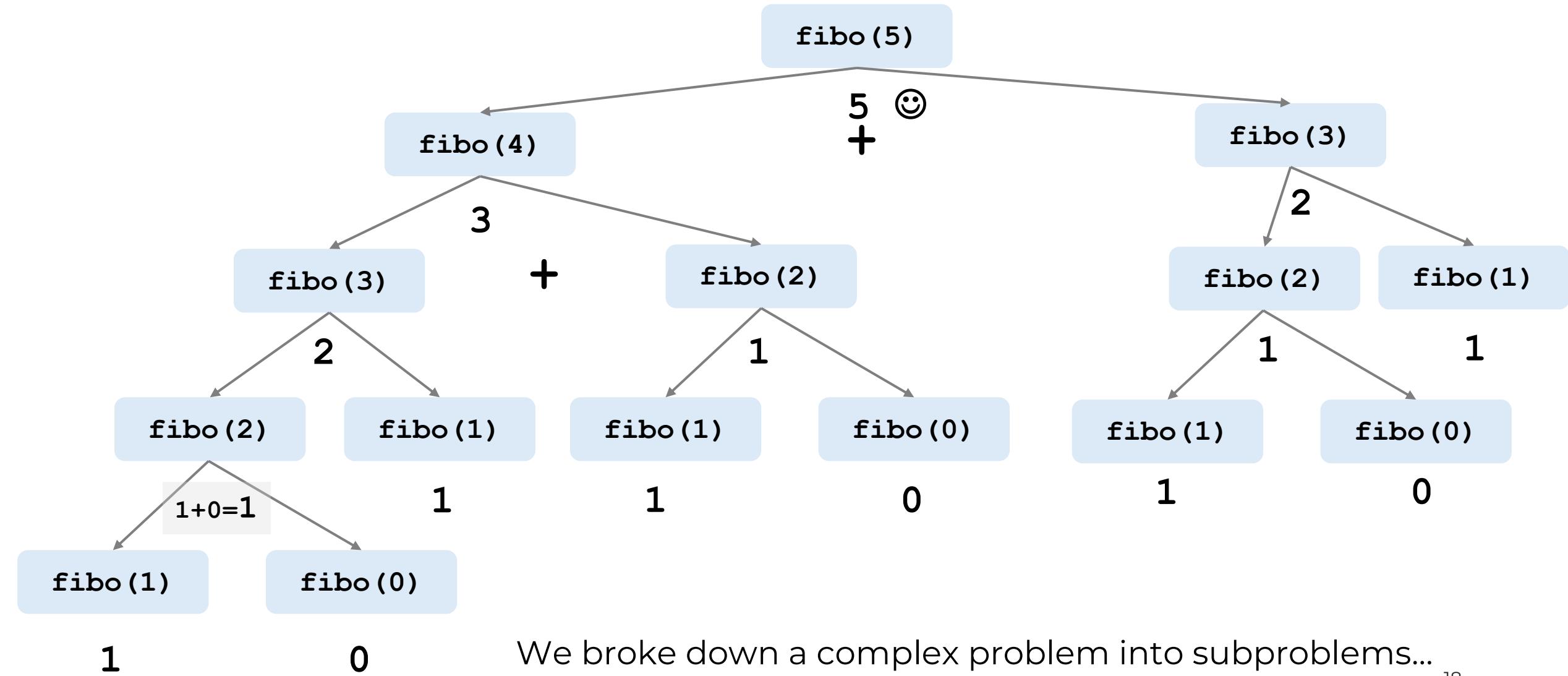
        return 0

    else:

        return n + recursiveSum (n - 1)
```

Let's see in practice!

We solve it recursively!

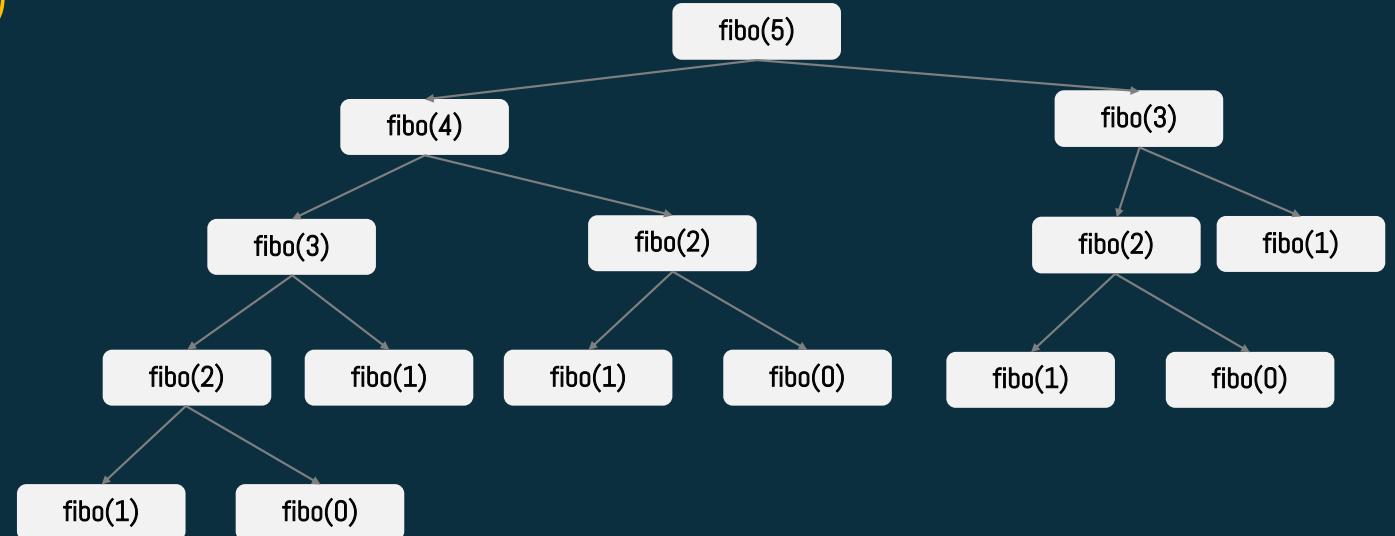


Quiz!

- ▶ What about computational complexity?
 - What is the time complexity of the Fibonacci method we just explored?
 - Answer: $O(2^n)$ (exponential)

- ▶ Is this good?

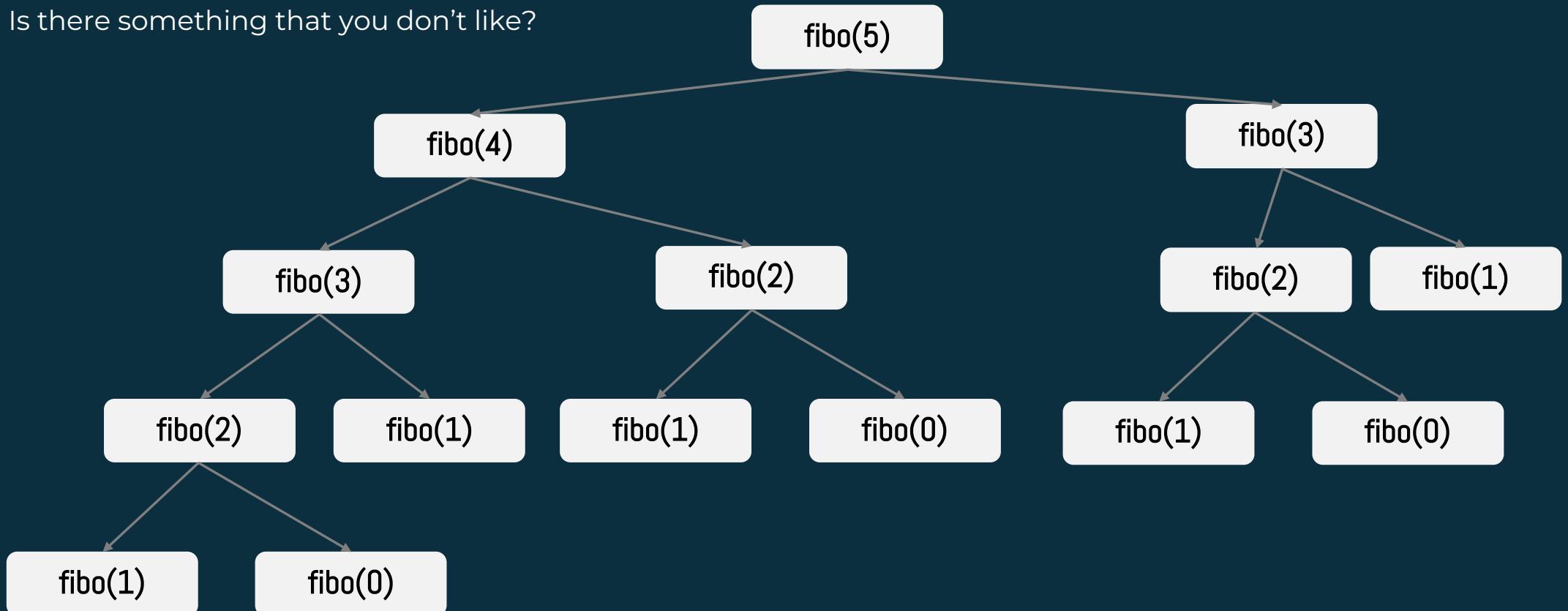
- Answer: $O(No!)$



Quiz!

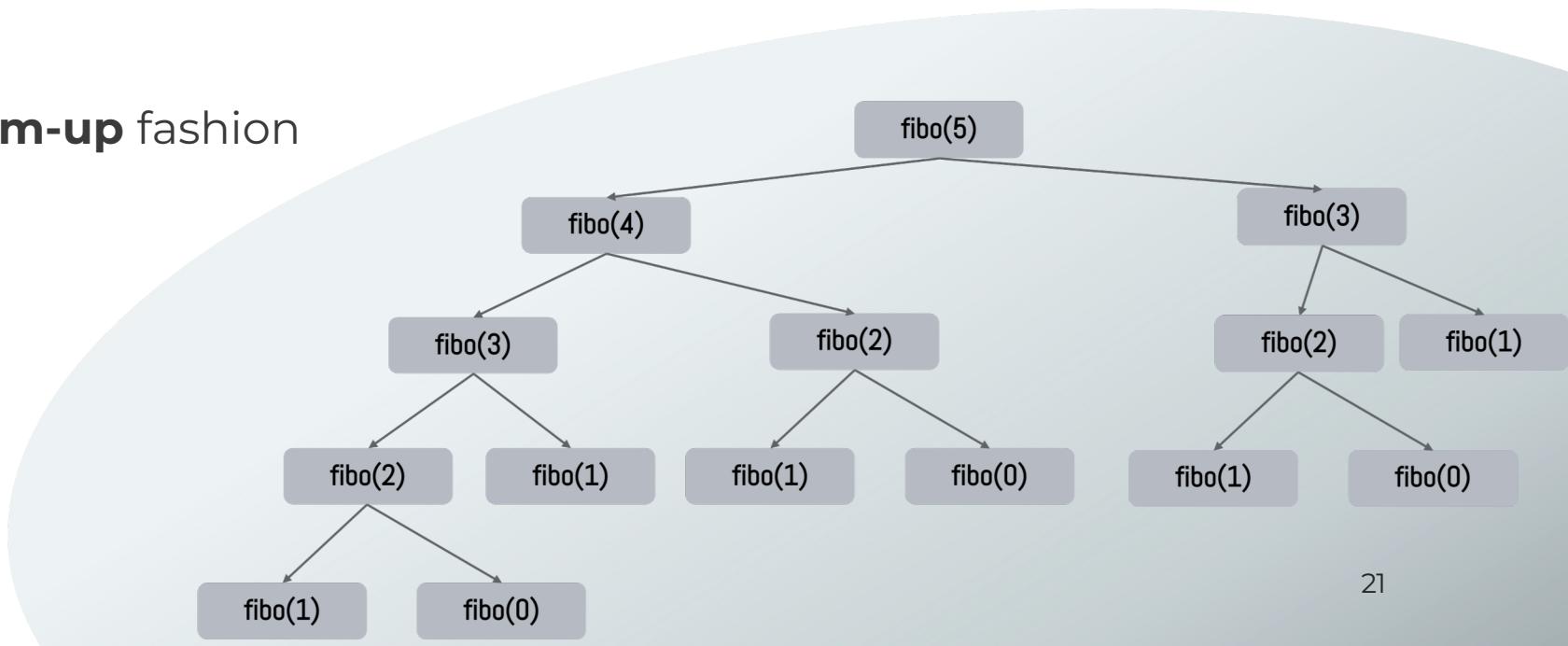
- ▶ Can you do better?

- Is there something that you don't like?



Can we do it better?

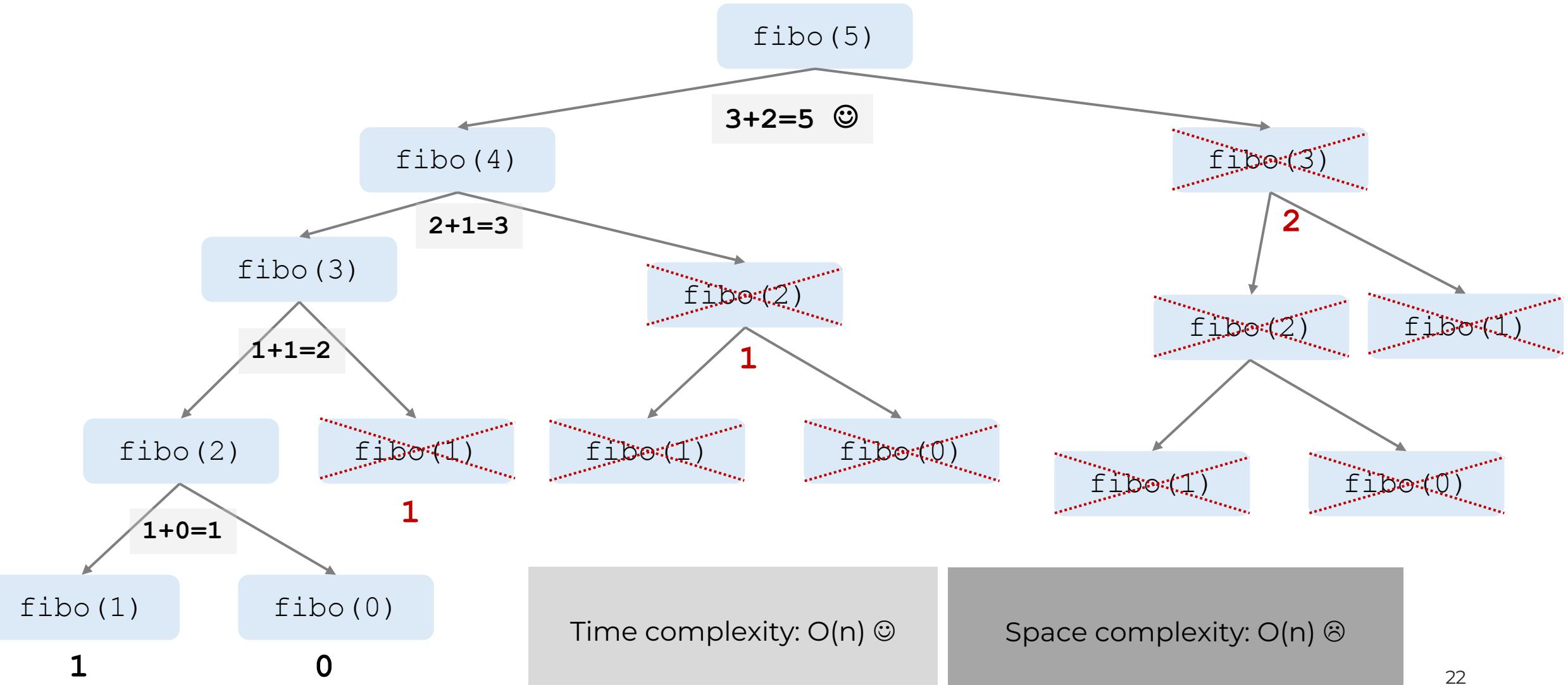
- ▶ **What if we avoid calculating the same Fibonacci numbers again and again?**
- ▶ What if we memorise the results of an existing calculation and use it in the next iteration?
 - Let's solve it in a **bottom-up** fashion



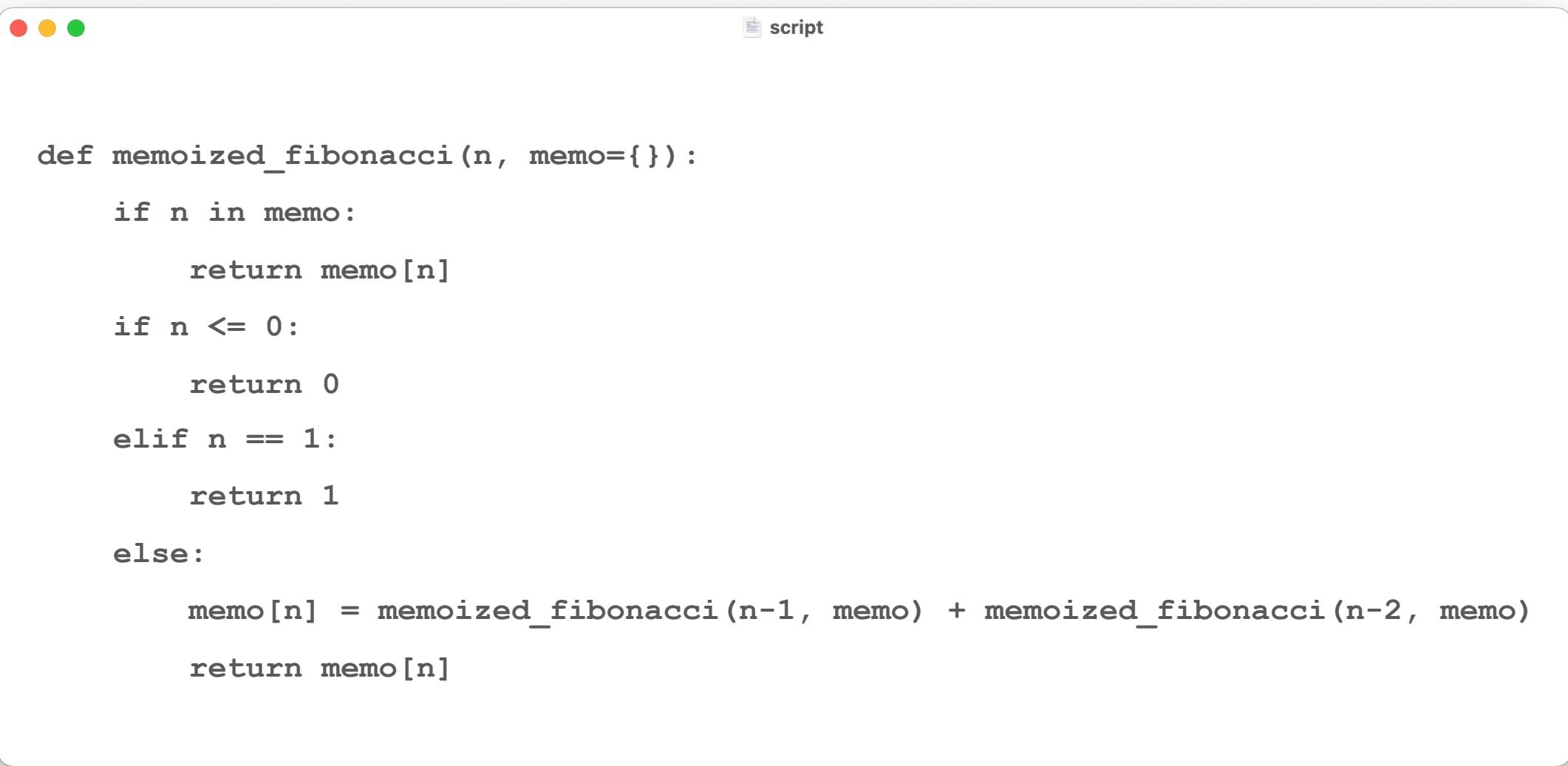
0	1	1	2	3	5
---	---	---	---	---	---

Let's use dynamic programming!

0 1 2 3 4 5



Using dynamic programming



A screenshot of a Mac OS X application window titled "script". The window contains Python code for calculating the nth Fibonacci number using dynamic programming with memoization. The code defines a function `memoized_fibonacci` that takes an integer `n` and an optional `memo` dictionary. It first checks if `n` is in the `memo` dictionary. If it is, it returns the value from `memo`. If `n` is less than or equal to 0, it returns 0. If `n` is 1, it returns 1. Otherwise, it calculates the value by summing the results of `memoized_fibonacci(n-1, memo)` and `memoized_fibonacci(n-2, memo)`, then stores the result in `memo[n]` before returning it.

```
def memoized_fibonacci(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        memo[n] = memoized_fibonacci(n-1, memo) + memoized_fibonacci(n-2, memo)
    return memo[n]
```

Summary

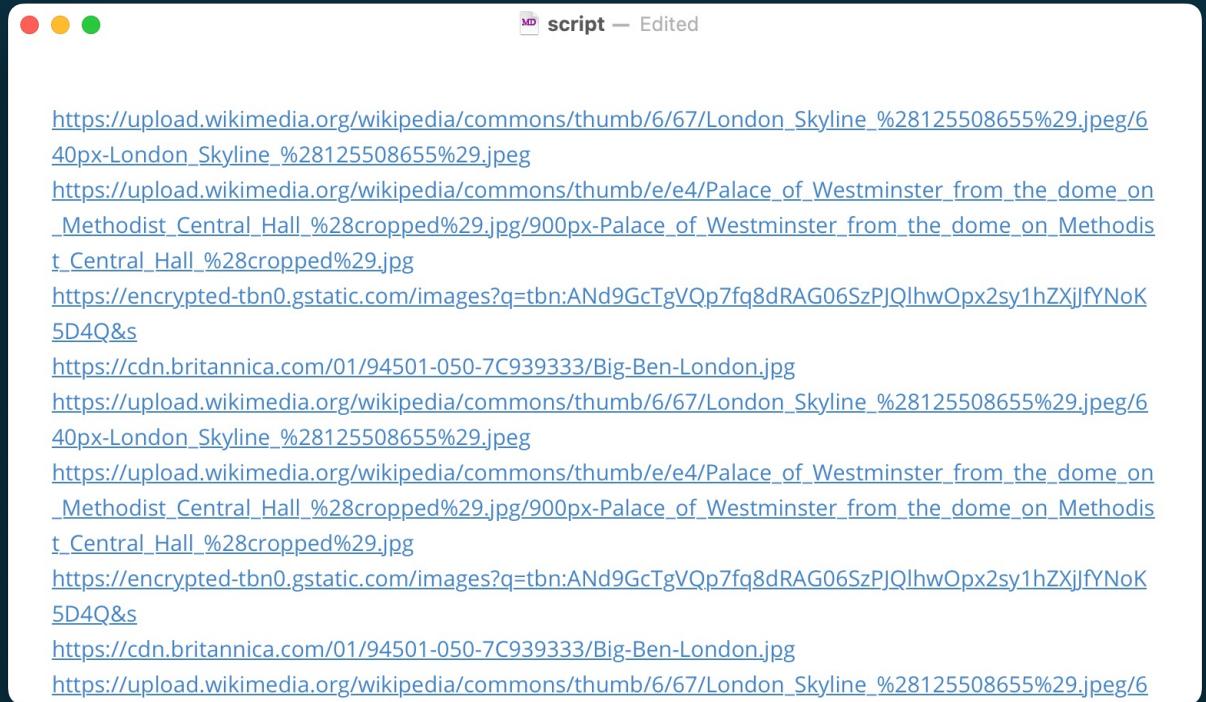
- ▶ **Divide and conquer** breaks down a problem into two or more sub-problems
 - Until these become simple enough to be solved directly.
- ▶ **Greedy methods** (top-down) solve a problem as quickly as possible.
 - Make whatever choice seems best now and then solve the sub-problems arising after.
- ▶ **Dynamic programming** (bottom-up) solves a problem as efficiently as possible.
 - Store the result for future purposes so you do not need to compute it again.

How do you avoid the complexity wall?



What is your plan?

- ▶ A text file with URLs of photos
- ▶ Task:
 - Download the photos.
- ▶ What is your strategy to approach this problem?



The screenshot shows a terminal window with the title bar "script — Edited". The window contains a list of URLs, likely from a text file, which are repeated multiple times. The URLs include:

- https://upload.wikimedia.org/wikipedia/commons/thumb/6/67/London_Skyline_%28125508655%29.jpeg/640px-London_Skyline_%28125508655%29.jpeg
- https://upload.wikimedia.org/wikipedia/commons/thumb/e/e4/Palace_of_Westminster_from_the_dome_on_Methodist_Central_Hall_%28cropped%29.jpg/900px-Palace_of_Westminster_from_the_dome_on_Methodist_Central_Hall_%28cropped%29.jpg
- <https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcTgVQp7fq8dRAG06SzPQlhwoPx2sy1hZXjjfYNoK5D4Q&s>
- <https://cdn.britannica.com/01/94501-050-7C939333/Big-Ben-London.jpg>
- https://upload.wikimedia.org/wikipedia/commons/thumb/6/67/London_Skyline_%28125508655%29.jpeg/640px-London_Skyline_%28125508655%29.jpeg
- https://upload.wikimedia.org/wikipedia/commons/thumb/e/e4/Palace_of_Westminster_from_the_dome_on_Methodist_Central_Hall_%28cropped%29.jpg/900px-Palace_of_Westminster_from_the_dome_on_Methodist_Central_Hall_%28cropped%29.jpg
- <https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcTgVQp7fq8dRAG06SzPQlhwoPx2sy1hZXjjfYNoK5D4Q&s>
- <https://cdn.britannica.com/01/94501-050-7C939333/Big-Ben-London.jpg>
- https://upload.wikimedia.org/wikipedia/commons/thumb/6/67/London_Skyline_%28125508655%29.jpeg/640px-London_Skyline_%28125508655%29.jpeg

Solutions

Serial

- ▶ Write a function to load text into a list:

`load_data(afile)`

- ▶ Write a function that downloads a photo from a given URL.

`get_img(aurl)`

- ▶ For each URL in the list, call the

`get_img(img)`

Parallel

- ▶ Write a function to load data into a list:

`load_data(afile)`

- ▶ Write a function that downloads a photo from a given URL.

`get_img(aurl)`

- ▶ Create a set of processes to call the

`get_img` using each `url`

What is your plan?

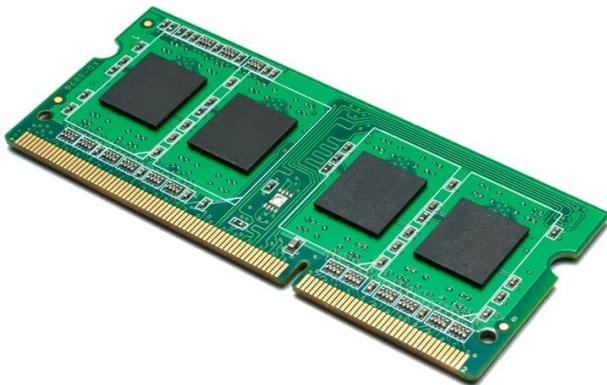
- ▶ A folder with 1000 YouTube videos...
- ▶ Task:
 - Analyse each video and determine if the content is suitable for children.
- ▶ What is your strategy to approach this problem?
- ▶ Write a function to load a video
 - `load_video(video_file)`
- ▶ Write a function to process a video.
 - `process_video(video)`
- ▶ Create a set of processes to run the `load_video()` in parallel.
 - How many processes?
 - \geq CPU cores
- ▶ Create a second set of processes to run the `process_video()` parallel.
 - CPU bound task

Intro to multiprocessing



Hard Disk (HD) memory

(Secondary memory)
Permanent storage



Random Access Memory (RAM)

(Main memory)
Temporary storage

Central Processing Unit (CPU)

The brain of a computer

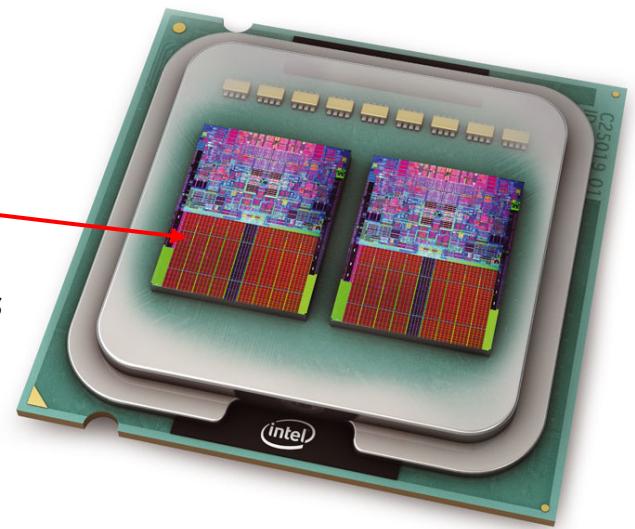


`print(10+20)`

*I live in the
RAM*

`num=10`

*Using one
of the cores*



Multicore processor

A CPU core is a single processing unit in the CPU that can execute instructions.

CPU

Inputs 5 3 6 2

2

6

3

5

```
def foo(n):  
    return n**n
```

Core 1

Core 2

Core 3

Core 4

Outputs 25 9 36 4

Example of serial processing

CPU

Inputs 5 3 6 2

2

5

3

6

Core 1

```
def foo(n):  
    return n**n
```

Core 2

```
def foo(n):  
    return n**n
```

Core 3

```
def foo(n):  
    return n**n
```

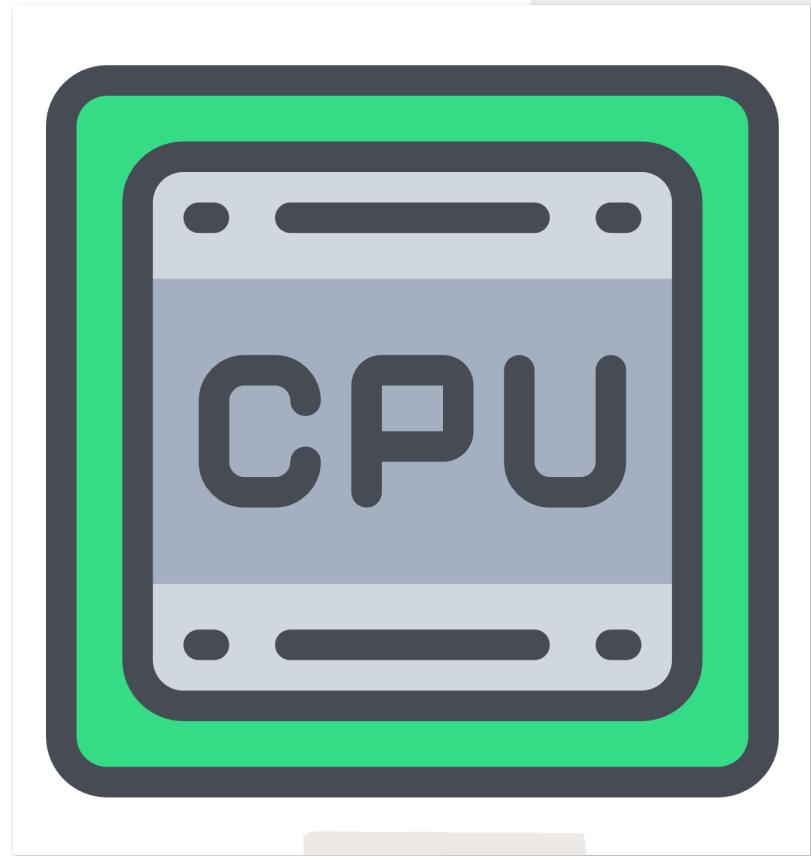
Core 4

Outputs 25 9 36 4

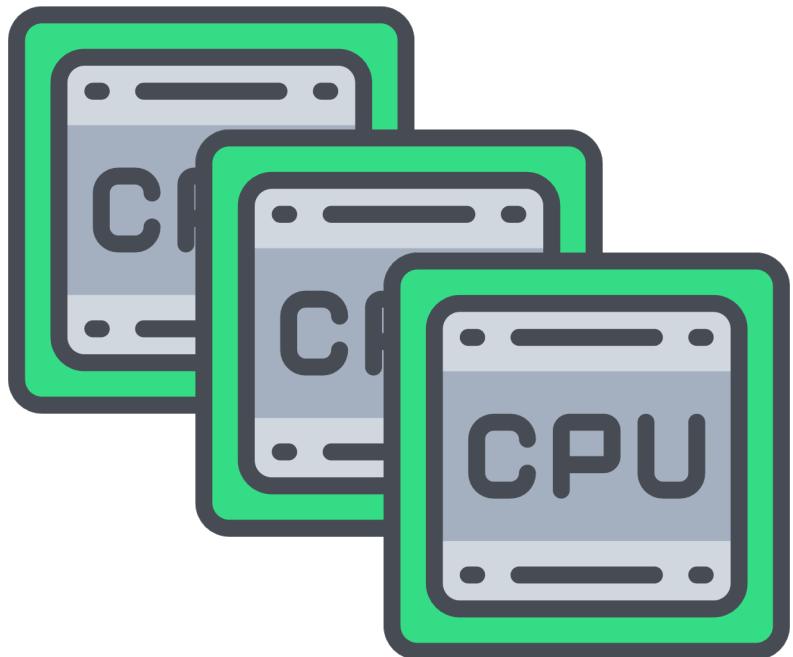
Example of parallel processing

Serial processing

- ▶ Executing instructions sequentially, one after another (linear).
 - Single processor use (1 core).
 - It is easier to implement and debug – tasks are completed in a strict order.
 - It may be slower for large tasks – it doesn't utilize multi-core processors.
- ▶ When to use?
 - Tasks must be done in a specific order – a task depends on the previous one.
 - The overhead of managing parallel tasks outweighs its benefits.



Parallel processing

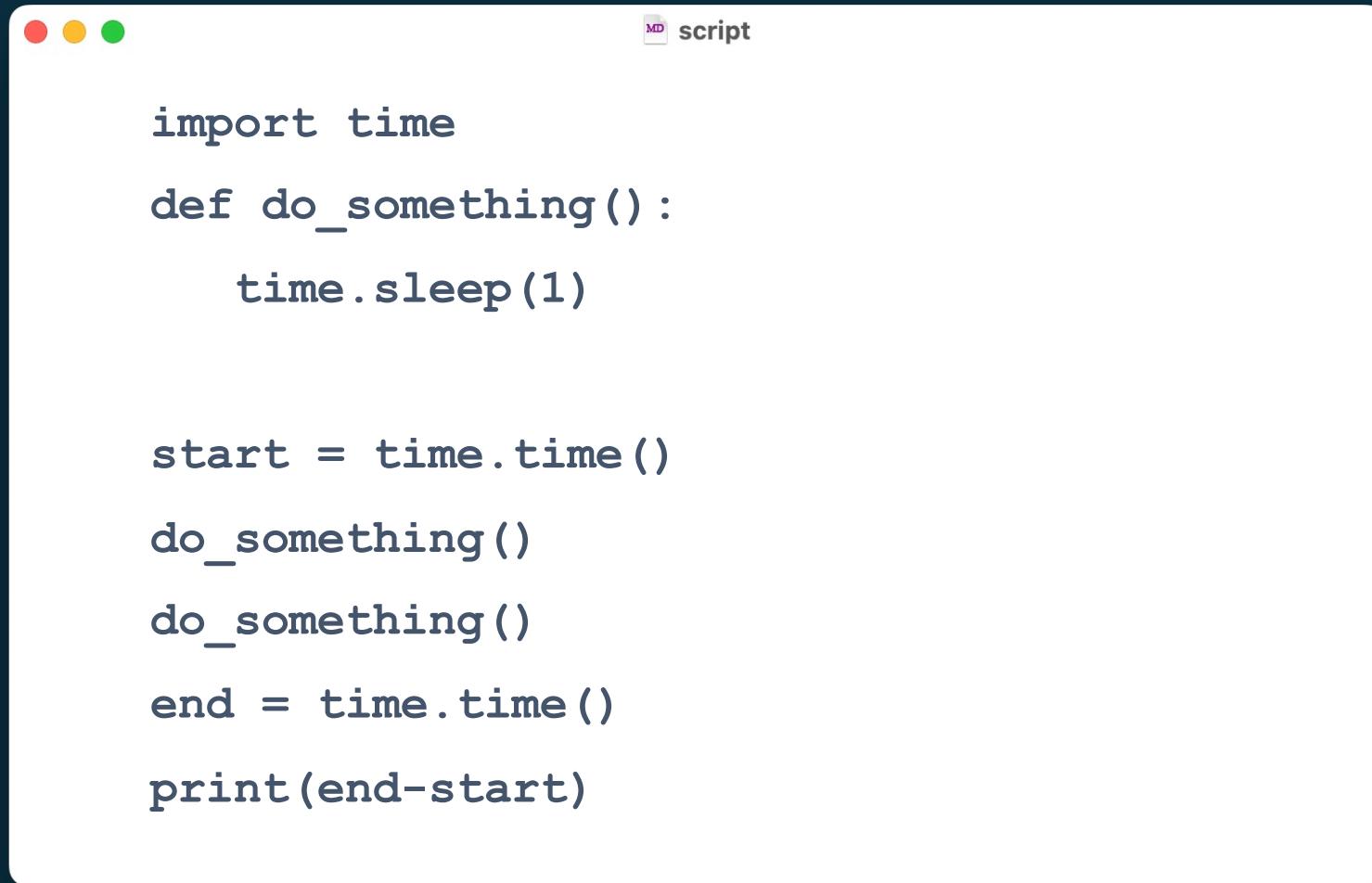


- ▶ Executing multiple tasks at the same time (concurrently) across different processors or cores.
 - Utilize multiple CPU cores in the same or different machines.
 - More complex to implement and debug due to concurrency issues:
 - Two or more processes access the same data.
 - Faster for large-scale processing tasks

When to use parallel?

- ▶ Large-scale data processing tasks
 - Big data analysis, scientific computing, and real-time processing.
- ▶ CPU-intensive applications like video encoding, large-scale simulations, or complex computations in physics and engineering.
- ▶ Applications that need to handle high volumes of simultaneous requests, such as web servers and database management systems.

How long does this script take to run?



A screenshot of a macOS terminal window titled "script". The window has the standard red, yellow, and green close buttons in the top-left corner. The title bar shows the word "script" next to a small icon. The terminal window contains the following Python code:

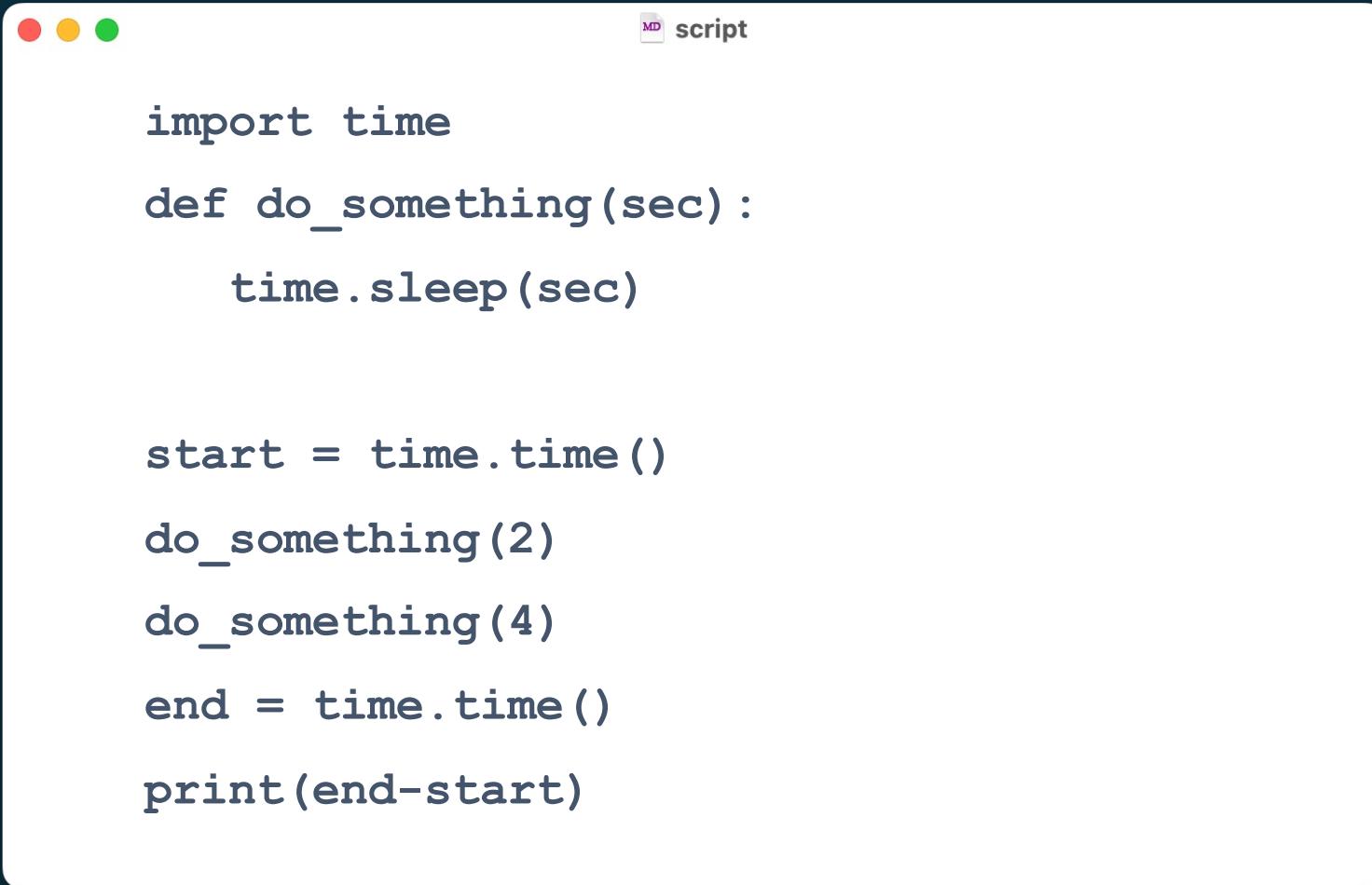
```
import time

def do_something():
    time.sleep(1)

start = time.time()
do_something()
do_something()
end = time.time()
print(end-start)
```

2 seconds!

What about this one?



A screenshot of a macOS terminal window. The window title is "script". The terminal contains the following Python code:

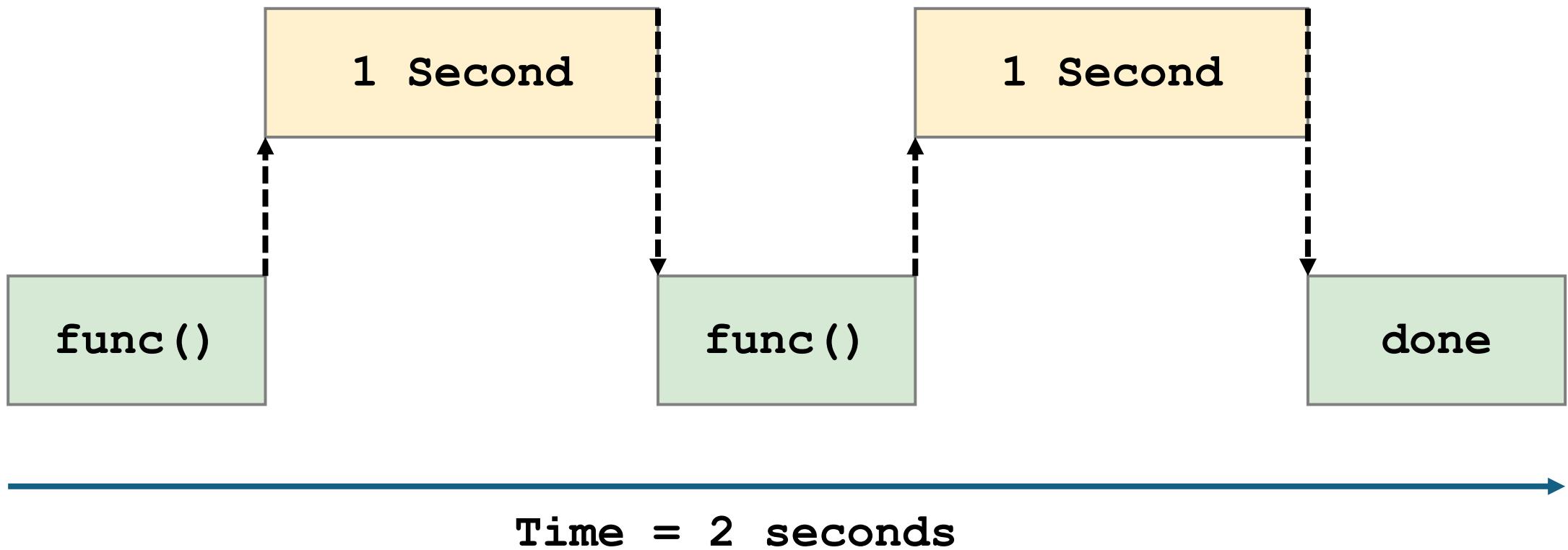
```
import time

def do_something(sec):
    time.sleep(sec)

start = time.time()
do_something(2)
do_something(4)
end = time.time()
print(end-start)
```

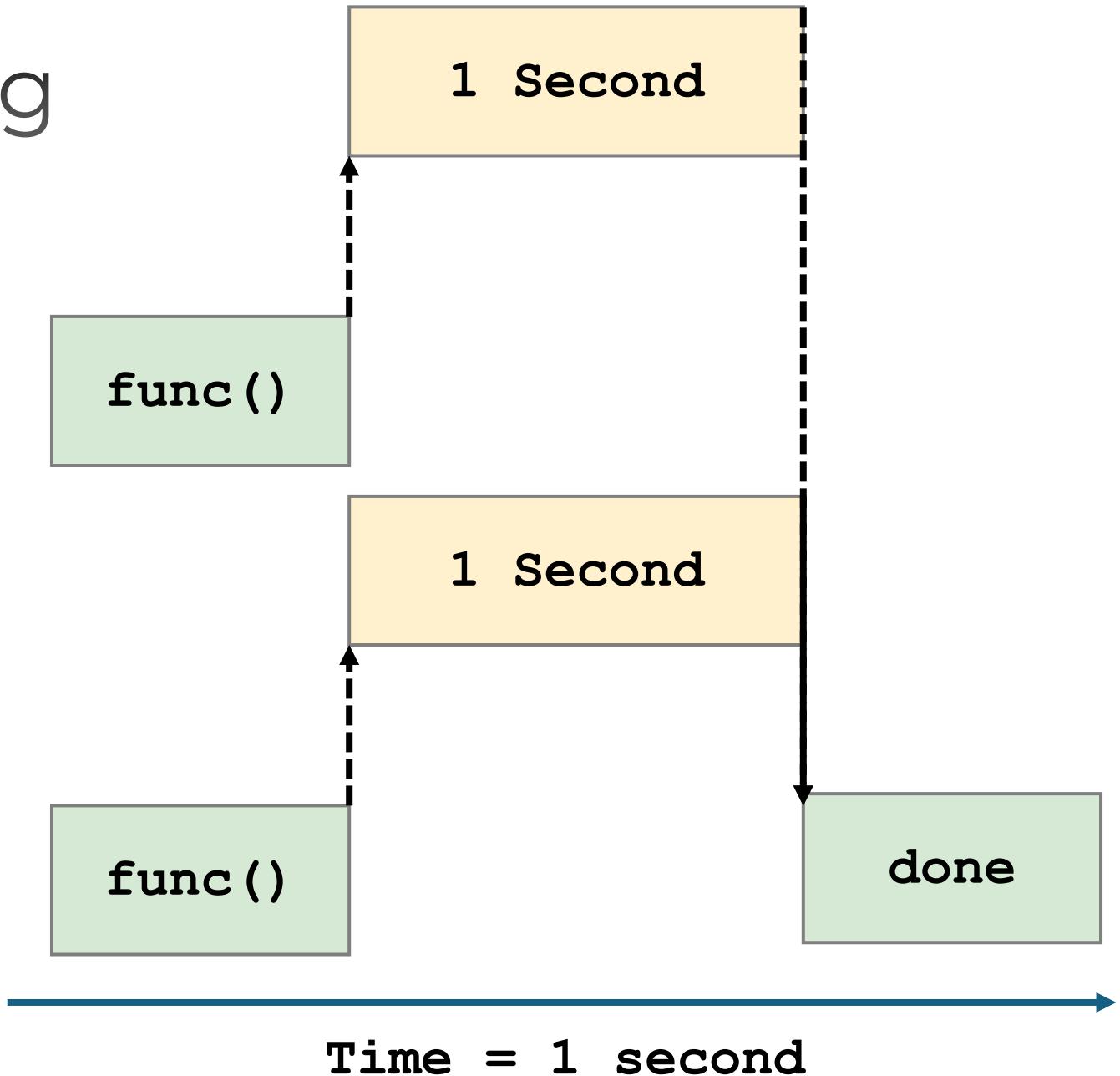
6 seconds!

Up until now, we have been inefficient!



Parallel processing

- ▶ “Processes” can run the operations on the data
- ▶ One piece at a time, then combine the results at the end to get the complete result.



Serial vs multi-processing vs multi-threading

- ▶ Serial processing:

`1 problem = 1 python program = 1 process = 1 task`

- ▶ Multi-processing:

`1 problem = 1 python program = n processes = n tasks`

- ▶ Multi-threading:

`1 problem = 1 python program = 1 process = n threads = n tasks`

(threads share memory)

What is a process?

- ▶ **A process refers to an instance of a running program.**
 - It is a self-contained execution environment with its own memory space, meaning it runs independently of other processes.
- ▶ Python **multiprocessing** module allows you to create and manage new processes.

Library:

```
import multiprocessing
```

Process types

► **CPU-bound**

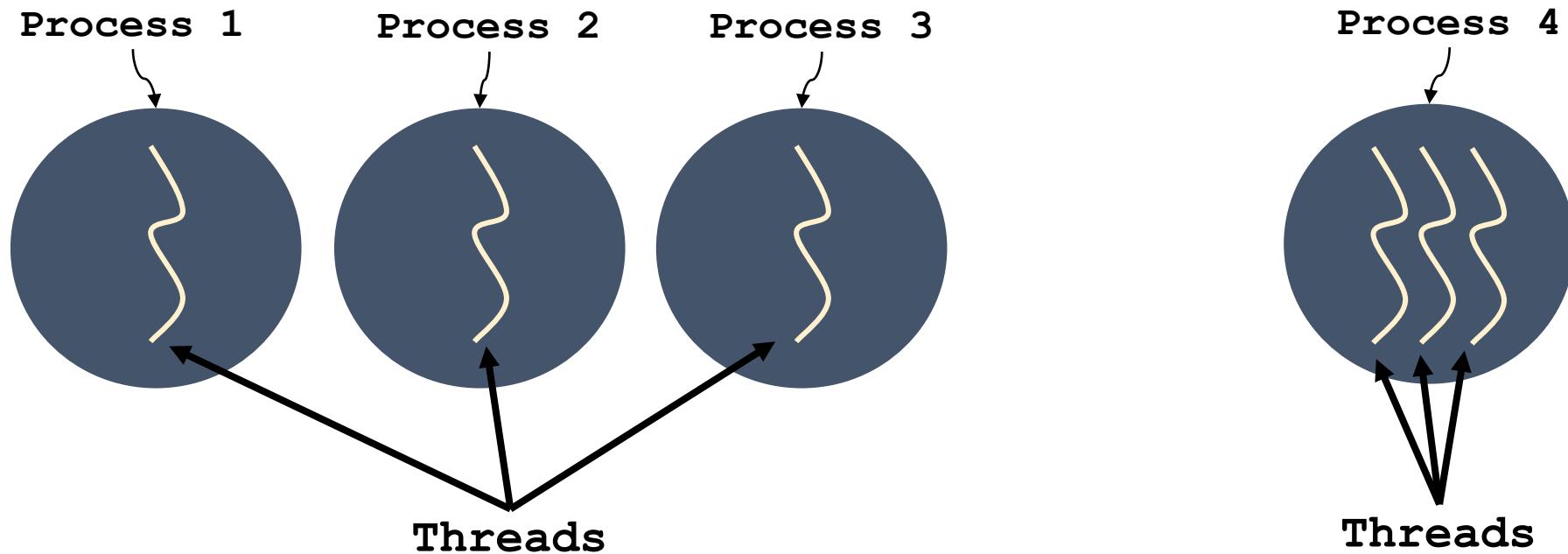
- Requiring significant computation – spends most of its time utilizing the CPU
- Reduce computation time or distribute the load across multiple CPU cores

► **Input/Output (I/O)-bound**

- Requiring significant computation – spends most of its time utilizing the disk.
- The bottleneck is the speed at which the data is received or sent.

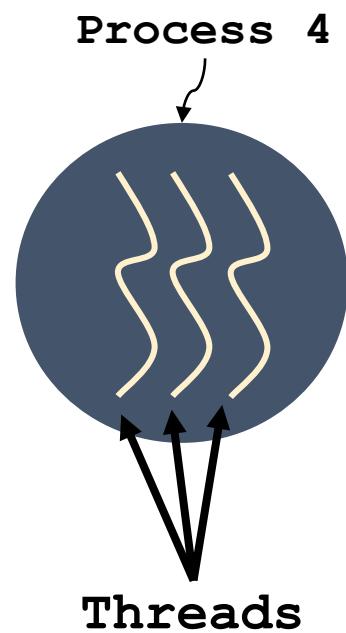
Processes and threads...

- ▶ Threads live inside “processes” and share memory!



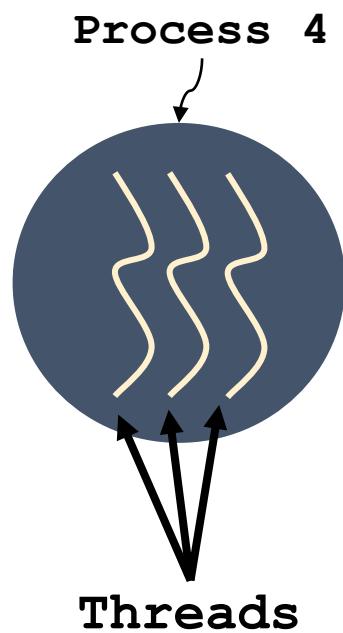
Why to use threads?

- ▶ Allow a single application to do many things at once
- ▶ Example
 - Thread to collect weather data
 - Thread to analyze weather trends
 - Thread to update forecasts
- ▶ Ideal for I/O operations!



Threads

- ▶ Threads are components of a process that can run in parallel.
- ▶ Multiple threads can be in a process and share the same memory space.
 - All threads share the variables declared within the program.



Processes vs Threads

Processes

- ▶ An instance of a program that is being executed
- ▶ Processes operate independently
- ▶ They do not share memory space with other processes directly
- ▶ **Ideal for CPU-bound tasks**

Threads

- ▶ The smallest unit of processing
- ▶ Threads in the same process share the same memory
- ▶ Less resource-intensive than creating a process
- ▶ **Ideal for I/O bound tasks**

Parallel

Serial

```
import time

start = time.perf_counter()

def do_something():
    time.sleep(1)

do_something()
do_something()

finish = time.perf_counter()

print(f'Finished in {finish-start} seconds(s) ')
```

```
script — Edited

import multiprocessing
import time

start = time.perf_counter()

def do_something():
    time.sleep(1)

p1 = multiprocessing.Process(target=do_something)
p2 = multiprocessing.Process(target=do_something)

p1.start()
p2.start()

p1.join()
p2.join()

finish = time.perf_counter()

print(f'Finished in {finish-start} seconds(s) ')
```

Multiprocessing tutorial

- ▶ Let's run it together!

- Using the Visual Studio Code

The screenshot shows a Visual Studio Code interface with a search bar at the top containing "bda". The left sidebar lists files in a "tutorial" folder, including "tutorial1.py", "tutorial2.py", "tutorial3.py", "tutorial4.py", "tutorial5.py", and "tutorial6.md". The main editor area displays the content of "tutorial1.py". The terminal at the bottom shows the output of running the script.

```
... tutorial1.py ...
session3 > tutorial > tutorial1.py
1 """
2 Tutorial 1 Script Overview:
3
4 This script demonstrates the difference in performance and execution methodology between s
5
6 Features:
7 * Serial Execution: Runs a set of tasks sequentially, one after the other, showing the tot
8 * Multiprocessing Execution: Utilizes the multiprocessing. Process class to run the same s
9
10 The script provides examples of:
11 * Setting up and starting multiprocessing.Process instances manually.
12 * Synchronizing tasks using the start() and join() methods to ensure that the main program
13
14 Usage:
15 * Run the script directly from the command line to see the output of serial vs. multiproce
16 * Adjust the number of tasks or complexity of tasks to see how multiprocessing can scale c
17 """
18
19 import time
20 import multiprocessing
21
22 def foo():
23     print("In foo...")
24     print("Running...")
```

PROBLEMS OUTPUT TERMINAL PORTS

> < TERMINAL

```
Counting objects: 100% (15/15), done.
Delta compression using up to 10 threads
Compressing objects: 100% (8/8), done.
Writing objects: 100% (8/8), 2.59 KiB | 2.59 MiB/s, done.
Total 8 (delta 4), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (4/4), completed with 4 local objects.
To https://github.com/warestack/bda.git
  01bb011..57a9361 main -> main
stelios@Stelios-MBP exercises %
```

Ln 18, Col 1 Spaces: 4 UTF-8 LF Python Go Live

Multiprocessing library

`multiprocessing.Process`

What is it?

- ▶ The **multiprocessing.Process** class allows you to create, launch, and manage new processes in parallel.
- ▶ Each **Process** object runs in its memory space, operating independently of other processes.

When to use it?

- ▶ Large-scale data processing:
 - Tasks can be easily parallelized and executed concurrently as batch jobs on chunks of data.
- ▶ Simulations:
 - Simulations or models that are computationally intensive and independent from each other.
- ▶ Batch Image or Video Processing:
 - Processing large sets of media files where each file can be processed independently.



```
from multiprocessing import Process

def my_function(args):

    print("Doing something with:", args)

if __name__ == '__main__':

    # Create a Process object that will run my_function

    p = Process(target=my_function, args=('some data',))

    p.start() # Start the process

    p.join() # Wait for the process to finish
```

Concurrent futures library

`concurrent.futures.ProcessPoolExecutor`

What is it?

- ▶ The **ProcessPoolExecutor** uses a pool of processes to execute function calls asynchronously and is designed to handle tasks that are CPU-bound.
- ▶ This approach is useful for parallel execution of tasks that can run independently and benefit from multiple cores.

When to use it?

- ▶ CPU-intensive Tasks:
 - Ideal for tasks that require heavy computation and can be distributed over multiple cores.
- ▶ Parallel Data Processing:
 - Useful in scenarios where a large amount of data needs to be processed in parallel, such as in data analysis or scientific computing.
- ▶ Complex Operations in Web Servers:
 - Can be used to handle CPU-heavy requests in web applications, running them in a background process pool to avoid blocking web server threads.



```
import concurrent.futures

import math


def compute_sqrt(number):

    result = math.sqrt(number)

    print(f"The square root of {number} is {result}")

    return result


if __name__ == "__main__":

    numbers = [25, 36, 49, 64, 81]

    with concurrent.futures.ProcessPoolExecutor() as executor:

        results = list(executor.map(compute_sqrt, numbers))
```

Pool library

pool

What is it?

- ▶ A powerful tool for parallel programming, enabling you to distribute tasks across multiple processors for concurrent execution.
- ▶ This is particularly useful for CPU-intensive operations that can benefit significantly from parallelization.
- ▶ The Pool class is designed to manage multiple worker processes, handling the dispatching of tasks and the collection of results seamlessly.

When to use it?

- ▶ Data Processing:
 - When you have a large dataset and need to perform computation-intensive operations on each element, multiprocessing.Pool() can significantly speed up processing by distributing the work across multiple CPUs.
- ▶ Simulations and Modeling:
 - For simulations that require a lot of independent computations, using Pool can reduce overall runtime.
- ▶ Batch Jobs:
 - Useful in scenarios where tasks are independent and can be completed in parallel, such as batch processing of files, images, or logs.



```
from multiprocessing import Pool

def square(n):

    return n * n

if __name__ == "__main__":
    numbers = [1, 2, 3, 4, 5]

    with Pool(5) as p:

        results = p.map(square, numbers)

    print(results)
```

Thank you!

- ▶ The lab starts soon!
(404-405)

O(no!)



Lab 3

Big Data Analytics

Lab activities

- ▶ Complete lab 3 activities and exercises.
- ▶ Use your preferred python IDE.