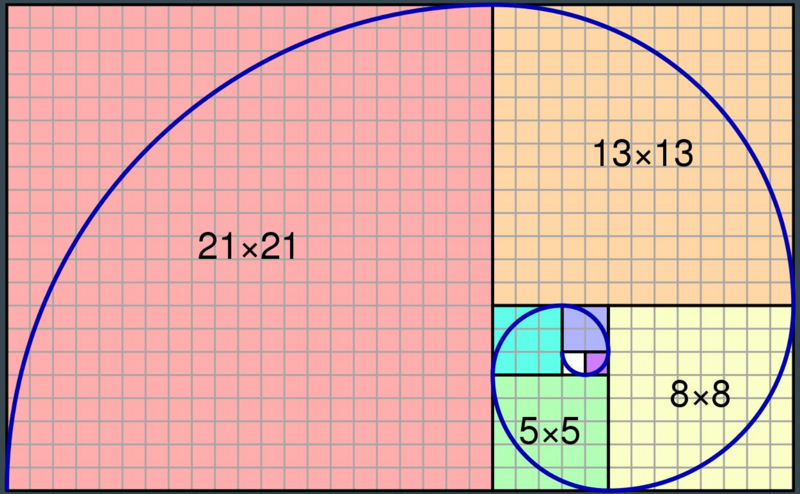
Computational complexity

Evaluating algorithms

An intuitive approach to complexity

Fibonacci sequence:

$$
\begin{cases}
F_0 = 0 \\
F_1 = 1 \\
F_n = F_{n-1} + F_{n-2} & \text{if } n \geq 2
\end{cases}
$$



Goal: find an algorithm to compute F_n

An intuitive approach to complexity

Two algorithms:

```
def fib(n):
    fs = [0, 1]
    for i in range(n-1):
        fs.append(fs[i] + fs[i+1])
    return fs[n]

n = int(input())
print(fib(n))
```

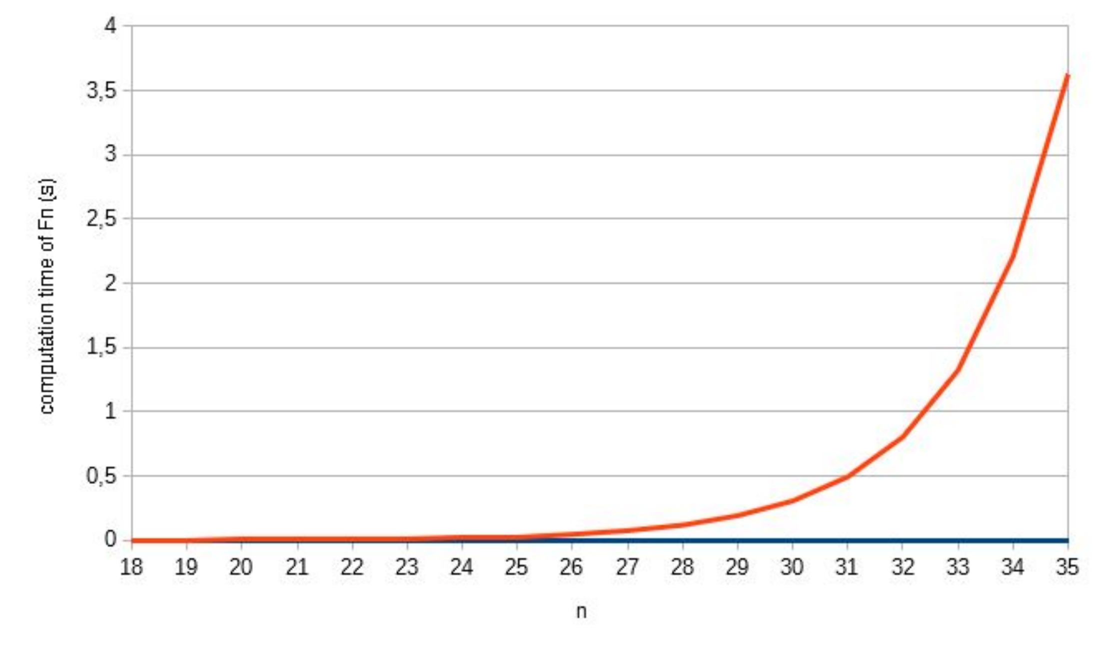
```
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)

n = int(input())
print(fib(n))
```

How efficient are they?

An intuitive approach to complexity

Let’s compare them with the module *time*:



In blue, the iterative algorithm

In orange, the naive recursive algorithm

What happened?

An intuitive approach to complexity

Two algorithms:

```
def fib(n):
     fs = [0, 1]
     for i in range(n-1):
         fs.append(fs[i] + fs[i+1])
     return fs[n]

n = int(input())
print(fib(n))
```

Loop of size n, with fixed operations inside → complexity is called *linear* in n

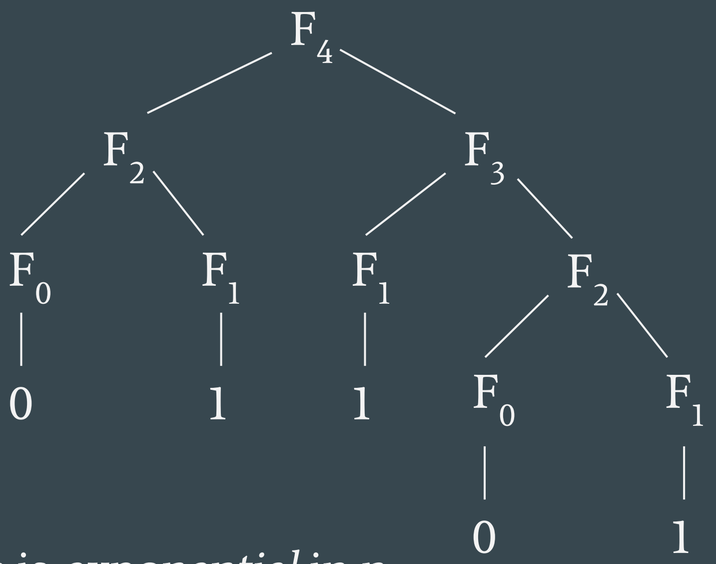
```
def fib(n):
     if n <= 1:
         return n
     else:
         return fib(n-1) + fib(n-2)

n = int(input())
print(fib(n))
```

Let's take a closer look at what happens with this algorithm...

An intuitive approach to complexity

Let’s execute the second algorithm for F_3:




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

n = int(input())
print(fib(n))
```

This is *exponential* in n

A more formal definition

- What we have evaluated previously is the *time complexity*, i.e the amount of time that it takes to run an algorithm, in function of its parameters
- When comparing memory consumption, we talk about *space complexity*

A more formal definition

When evaluating complexity, we use the *big O notation*:

$f(x) = O(g(x))$
if and only if there is x_0 and M such that:

$$|f(x)| \leq M\, g(x) \qquad \text{for all } x > x_0$$

For example:

- $n^2 - 3n + 5 \quad = O(n^2)$
- $42 \quad = O(1)$
- $-7\, n \log_2 n \quad = O(n \log n)$

This notation allows to evaluate how the program behaves depending on the size of the parameters.

Examples of time complexity

Let n be the size of an array.

Here are the time complexities of a few operations:

- $O(\log n)$ find an element in a sorted array with binary search
- $O(n)$ explore all elements of the array
- $O(n \log n)$ sort the array with a merge sort
- $O(n^2)$ find all the couples of elements of the array
- $O(n!)$ find all the permutations of the array

etc

Average VS worst case complexity

Sometimes the time needed to run an algorithms varies for inputs of the same size.

For example, the complexity of the quick sort is:

- $O(n \log n)$ on average
- $O(n^2)$ in the worst case

When talking about complexity, we usually refer to the *worst case* complexity.

Common complexities

- 1 constant
- log n logarithmic
- n linear
- n log n linearithmic
- n^2 quadratic
- n^3 cubic
- n^k polynomial
- k^n exponential
- n! factorial

Now, some exercises :)

Equivalence time - complexity

total time = number of operations / operations per second

In python3 ...

total time = number of operations / 10^7 - 10^8

Why is it useful?

Problem's constraints : $0 \le N \le 10^2$, $0 \le C \le 10^3$, Time limit = 1s

$N*C$	=>	10^5 operations	=>	0.001s
N^2*C	=>	10^7 operations	=>	0.1s
$N*C^2$	=>	10^8 operations	=>	1s
$N!*C$	=>	∞? operations	=>	NO