

# Complexity



# Complexity

- Computational Complexity
- Complexity in Software Design



# Computational Complexity

**Investigation of efficiency:** Amount of resources needed for algorithms working on data structures

# Dimensions of Complexity

## Time Complexity

- Time to run an algorithm with given data size
- Often main focus of complexity analysis

## Space Complexity

- Memory required to run an algorithm with given data size

# Quiz

How many bytes are in a double temperature array on a  $0.1^\circ \times 0.1^\circ$  lat-lon grid and 100 vertical model levels, containing hourly data for one day?



# Quiz

How many bytes are in a double temperature array of 3600 x 1800 x 100 x 24 elements?

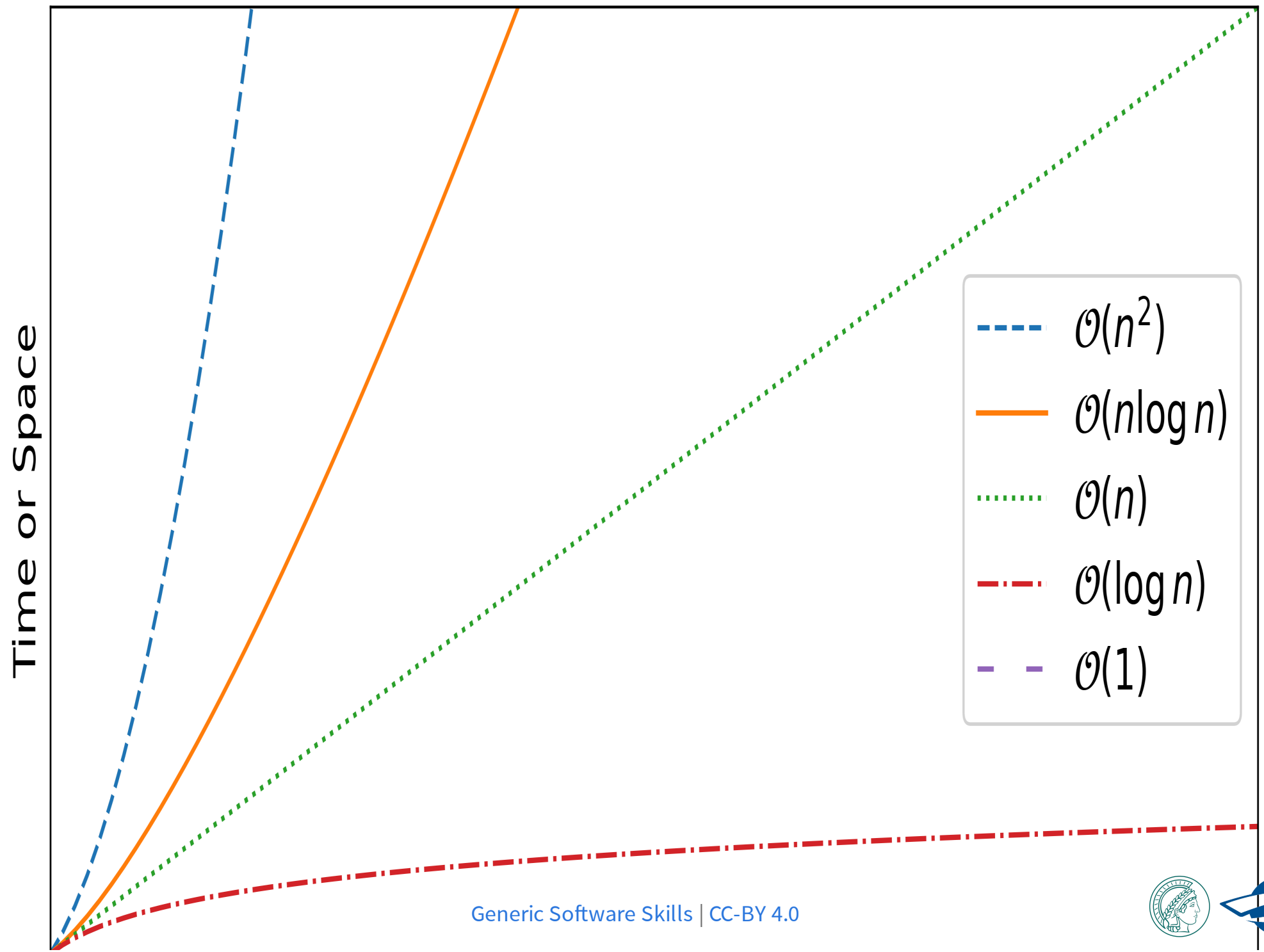
12441600000 or 12 GB  
(± overhead & compression)



# Big- $\mathcal{O}$ Notation

- growth in relation to input data size  $n$
- big- $\mathcal{O}$  notation describes asymptotic behaviour





# Big- $\mathcal{O}$ Notation

Order	Notation
constant	$\mathcal{O}(1)$
logarithmic	$\mathcal{O}(\log n)$
linear	$\mathcal{O}(n)$
quasilinear	$\mathcal{O}(n \log n)$
quadratic	$\mathcal{O}(n^2)$

# Quiz

How large is  $\log n$  on a computer?



# Quiz

How large is  $\log n$  on a computer?

64



# Quiz

How large is  $\log n$  on a computer?

64 (at least on a 64 bit machine)

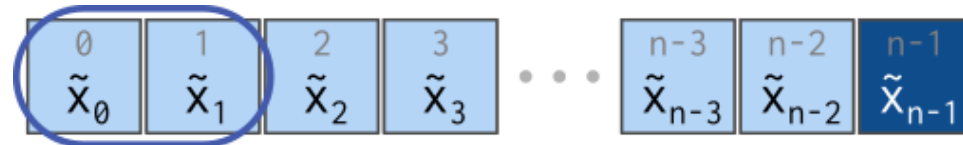


# Example

*sorting*



# Bubble Sort (1/2)



1. Array with  $n$  elements
2. Compare first two elements, swap them if left  $>$  right
3. Continue pairwise comparison until last elements
4. Keep repeating until everything is sorted

# Bubble Sort (2/2)

```
1 def swap_if_larger(values):
2     for i in range(len(values) - 1):
3         if values[i] > values[i+1]:
4             values[i], values[i+1] = values[i+1], values[i]
5             print(values)
```

```
1 def bubble_sort(values):
2     for _ in values:
3         swap_if_larger(values)
4
5 test_values = [6, 0, 3, 5, 1]
6 bubble_sort(test_values)
```

```
[0, 6, 3, 5, 1]
[0, 3, 6, 5, 1]
[0, 3, 5, 6, 1]
[0, 3, 5, 1, 6]
[0, 3, 1, 5, 6]
[0, 1, 3, 5, 6]
```



# Quick Sort (1/2)

CC-BY-SA [idea instructions](#)



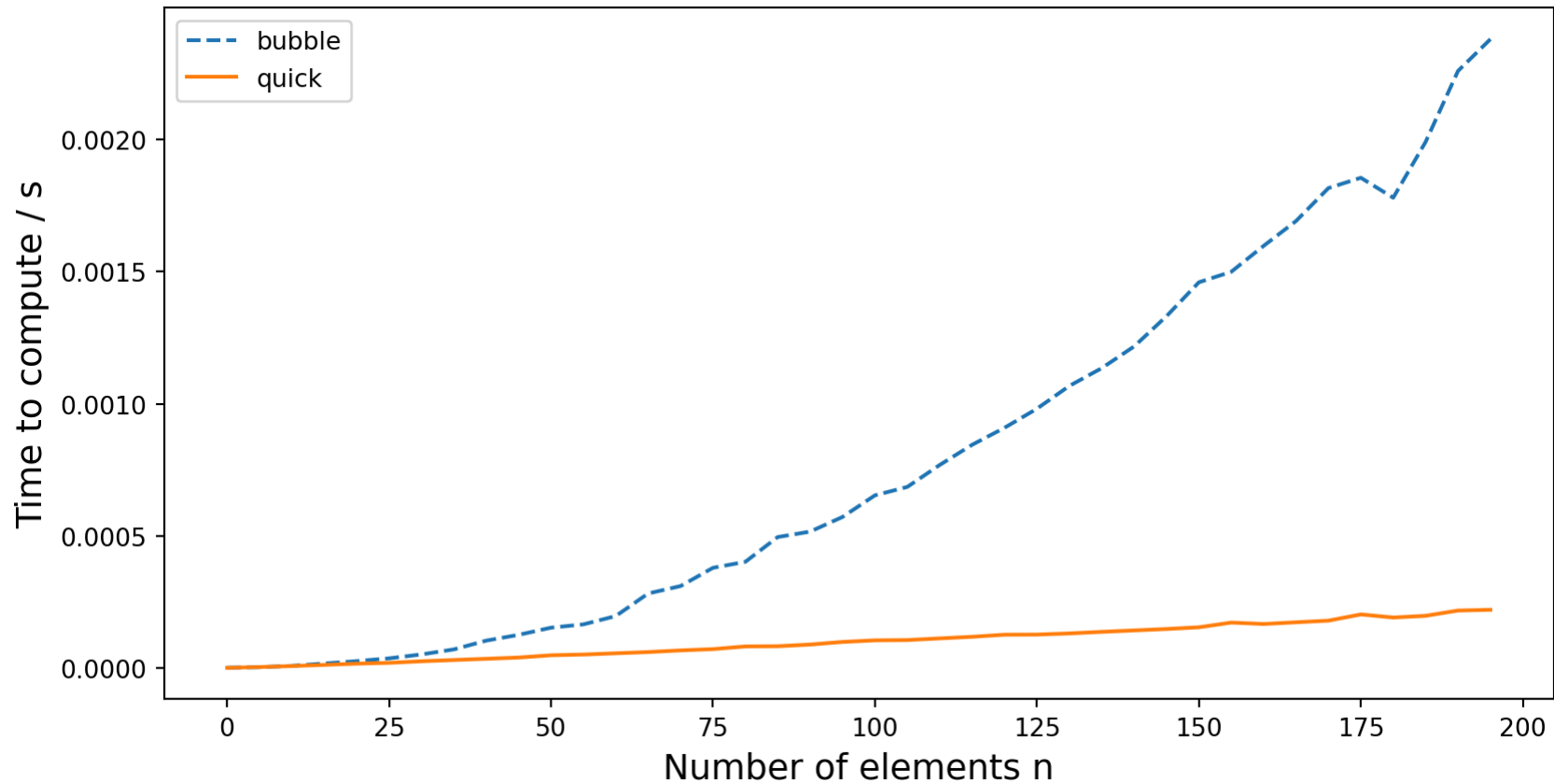
# Quick Sort (2/2)

```
1 def quicksort(l):
2     if len(l) == 0: return []
3     pivot = l[0]
4     lower = [e for e in l[1:] if e < pivot]
5     upper = [e for e in l[1:] if e >= pivot]
6     return quicksort(lower) + [pivot] + quicksort(upper)
7
8 test_values = [6, 0, 3, 1, 5, 7, 4, 2]
9 quicksort(test_values)
```

```
[0, 1, 2, 3, 4, 5, 6, 7]
```



# Bubble vs Quick



# There are more

You may want to study e.g.:

- Heapsort
- Mergesort
- Countingsort
- Bogosort

For an overview, [check Wikipedia](#)



# Example

*Fibonacci*



# Fibonacci series

$$f(0) = 0$$

$$f(1) = 1$$

$$f(n) = f(n - 2) + f(n - 1)$$

# Fibonacci series example (1/3)

## Direct implementation

```
1 def fib1(n):  
2     if n < 2: return n  
3     return fib1(n - 2) + fib1(n - 1)  
4  
5 [fib1(i) for i in range(10)]
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

```
1 %%time  
2 fib1(30)
```

```
CPU times: user 144 ms, sys: 0 ns, total: 144 ms
```

```
Wall time: 146 ms
```

```
832040
```



# Fibonacci series example (2/3)

## Iterative implementation with memory

```
1 def fib2(n):
2     if n < 2: return n
3     a = 0; b = 1
4     for i in range(2, n + 1): a, b = b, a + b
5     return b
6
7 [fib2(i) for i in range(10)]
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

```
1 %%time
2 fib2(30)
```

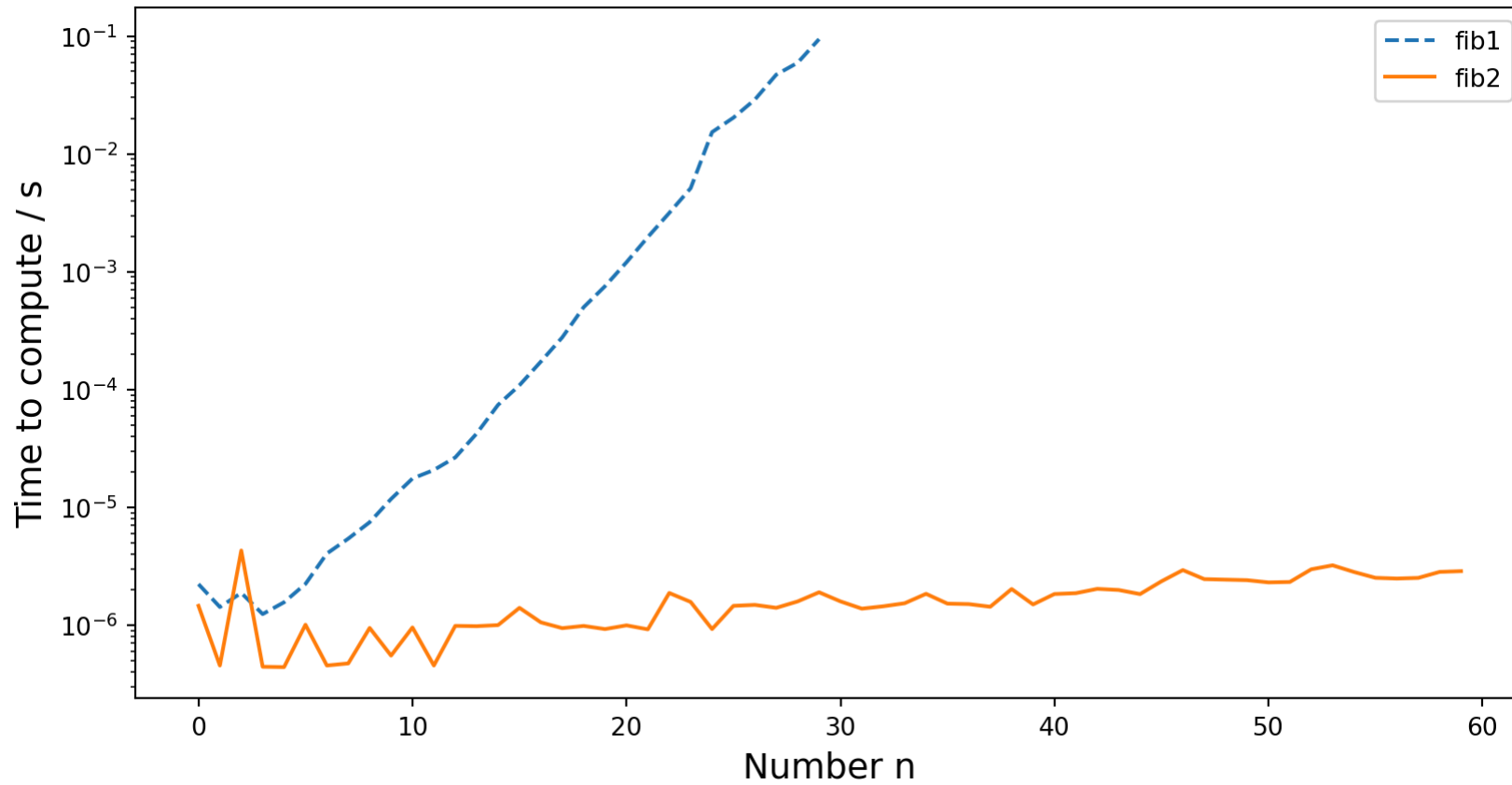
```
CPU times: user 6 µs, sys: 1 µs, total: 7 µs
```

```
Wall time: 10.7 µs
```

```
832040
```

 Substantially faster

# Performance Overview



# Fibonacci series example (3/3)

$$\begin{pmatrix} f_n \\ f_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} f_{n-1} \\ f_n \end{pmatrix}$$

$$\begin{pmatrix} f_n \\ f_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} f_0 \\ f_1 \end{pmatrix}$$



# vector

```
1 from dataclasses import dataclass
2
3 @texV # nicer display
4 @dataclass
5 class V2:
6     x: float; y: float
7     def __add__(s, o): # self, other
8         return V2(s.x + o.x, s.y + o.y)
9
10 V2(1, 2)
```

$$\begin{pmatrix} 1 \\ 2 \end{pmatrix}$$



# vector

$$1 \quad v_2(1, 2) + v_2(20, 40)$$

$$\begin{pmatrix} 21 \\ 42 \end{pmatrix}$$



# matrix

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} ax + by \\ cx + dy \end{pmatrix}$$

```

1 @texM # nicer display
2 @dataclass
3 class M2:
4     a: float; b: float; c: float; d: float
5
6     def __mul__(s, o): # self, other
7         if isinstance(o, V2):
8             return V2(s.a * o.x + s.b * o.y, s.c * o.x + s.d * o.y)
9         elif isinstance(o, M2):
10            return M2(s.a * o.a + s.b * o.c, s.a * o.b + s.b * o.d,
11                    s.c * o.a + s.d * o.c, s.c * o.b + s.d * o.d)

```



# test it:

```
1 m = M2(0, 1, 1, 1)
2 m
```

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

```
1 m * V2(0, 1)
```

$$\begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

```
1 m * m * V2(0, 1)
```

$$\begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

```
1 m * m * m * V2(0, 1)
```

$$\begin{pmatrix} 2 \\ 3 \end{pmatrix}$$

# power function

```
1 power(2, 4)
```

16

... applied on matrix class:

```
1 power(m, 3) * V2(0, 1)
```

$$\begin{pmatrix} 2 \\ 3 \end{pmatrix}$$

# Hands-on!

- Implement that generic power function.
- Do so with a time complexity better than  $\mathcal{O}(n)$



# fib3

```
1 def fib3(n):  
2     if n < 2: return n  
3     return (power(m, n) * V2(0, 1)).x  
4  
5 [fib3(i) for i in range(10)]
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

```
1 %%time  
2 fib3(30)
```

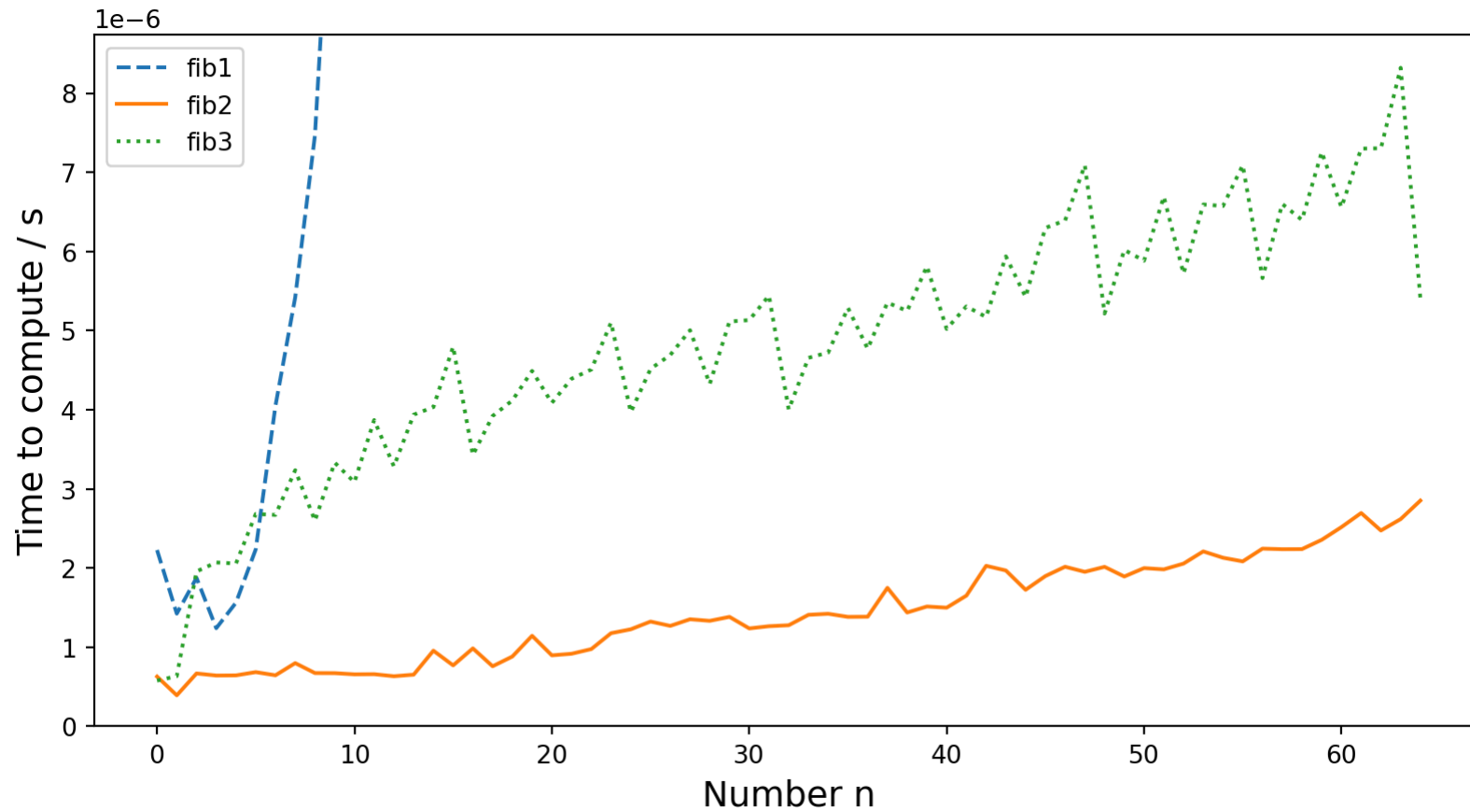
```
CPU times: user 21 µs, sys: 2 µs, total: 23 µs
```

```
Wall time: 26 µs
```

```
832040
```

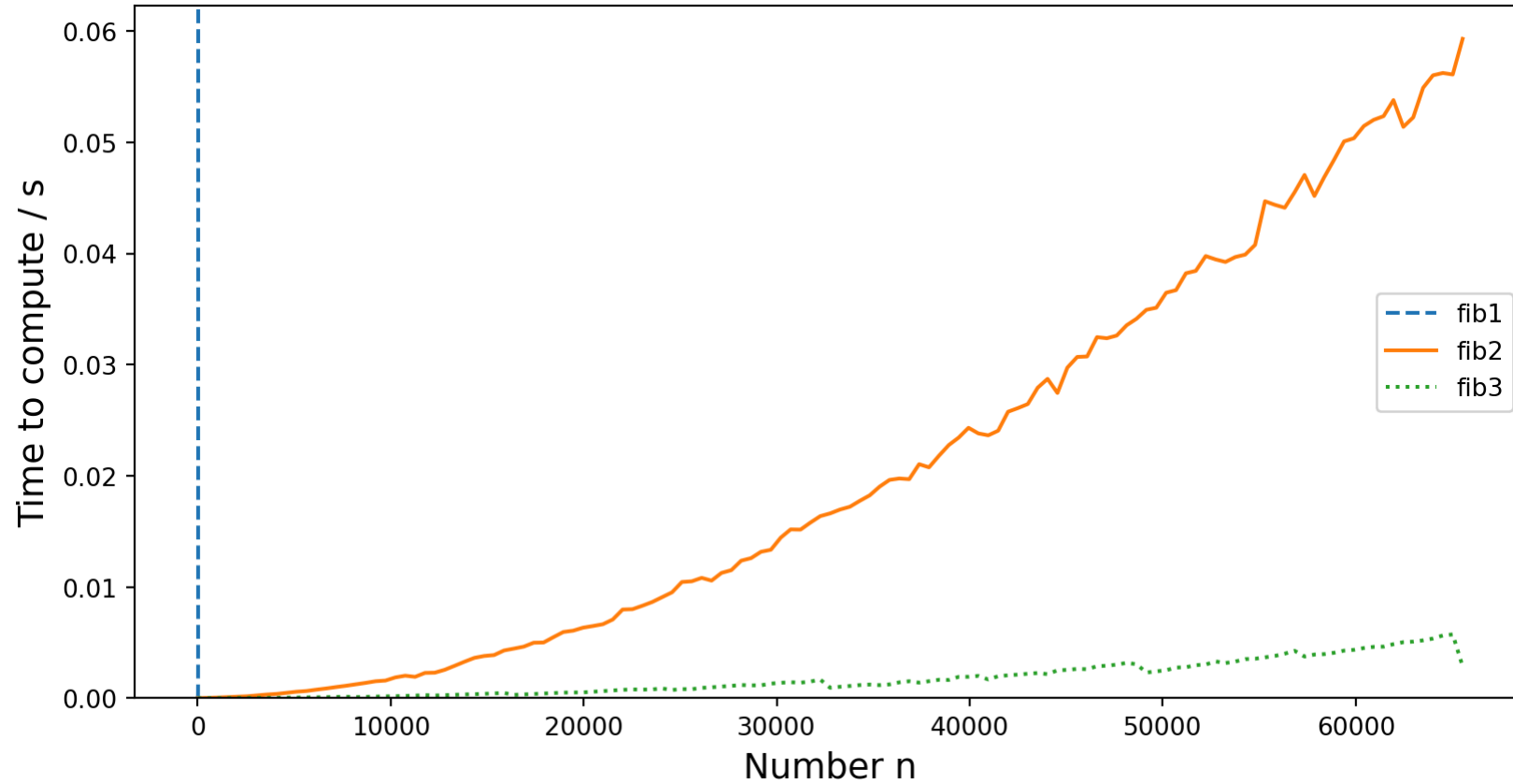


# Performance Overview



👉 Unfortunately slower 😬

# Performance Overview



👉 Faster for **large numbers**

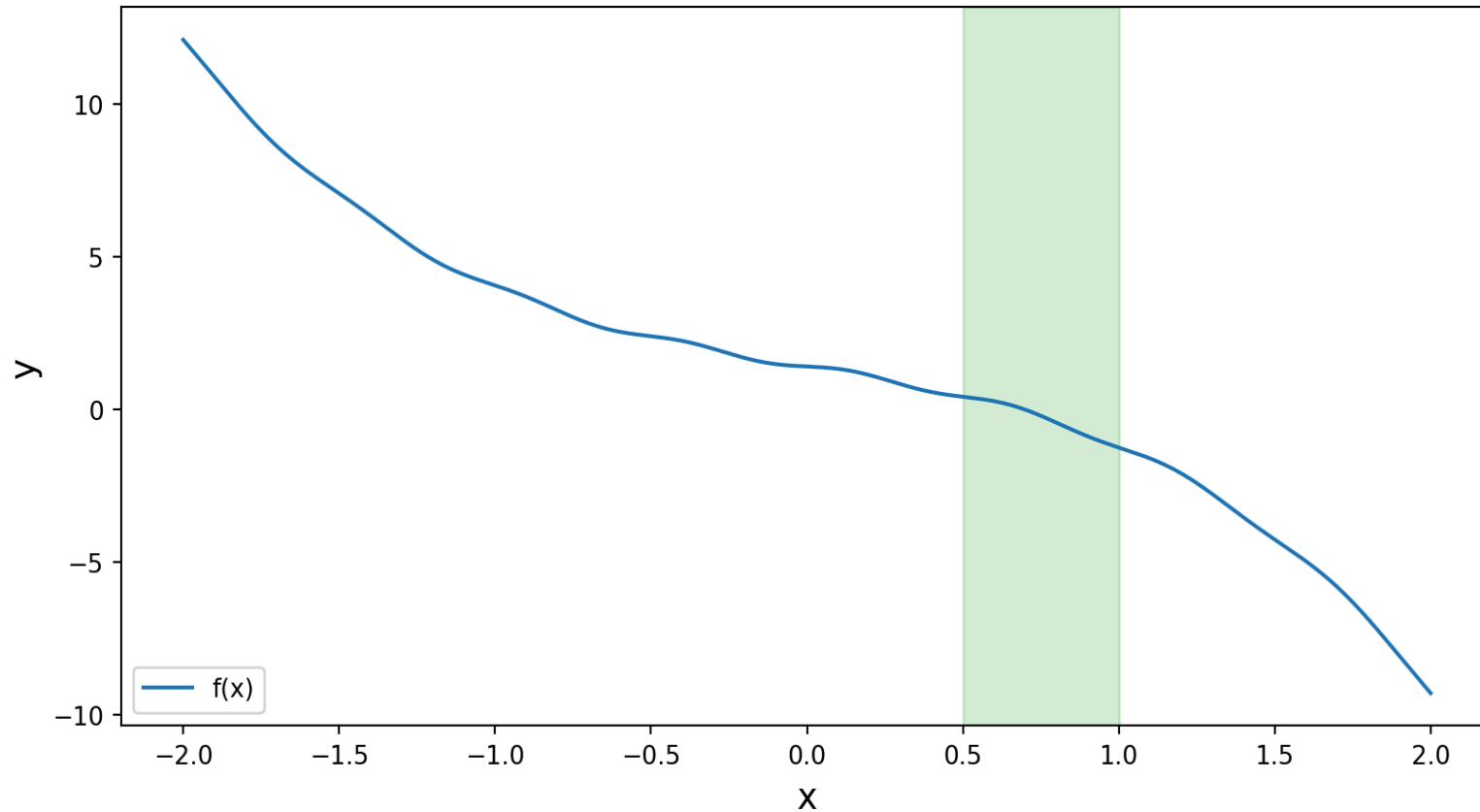
# Example

*hands-on*



# Hands-on example

$$f(x) = -0.9x^3 - 1.7x + 0.1 \sin(12x) + 1.4$$



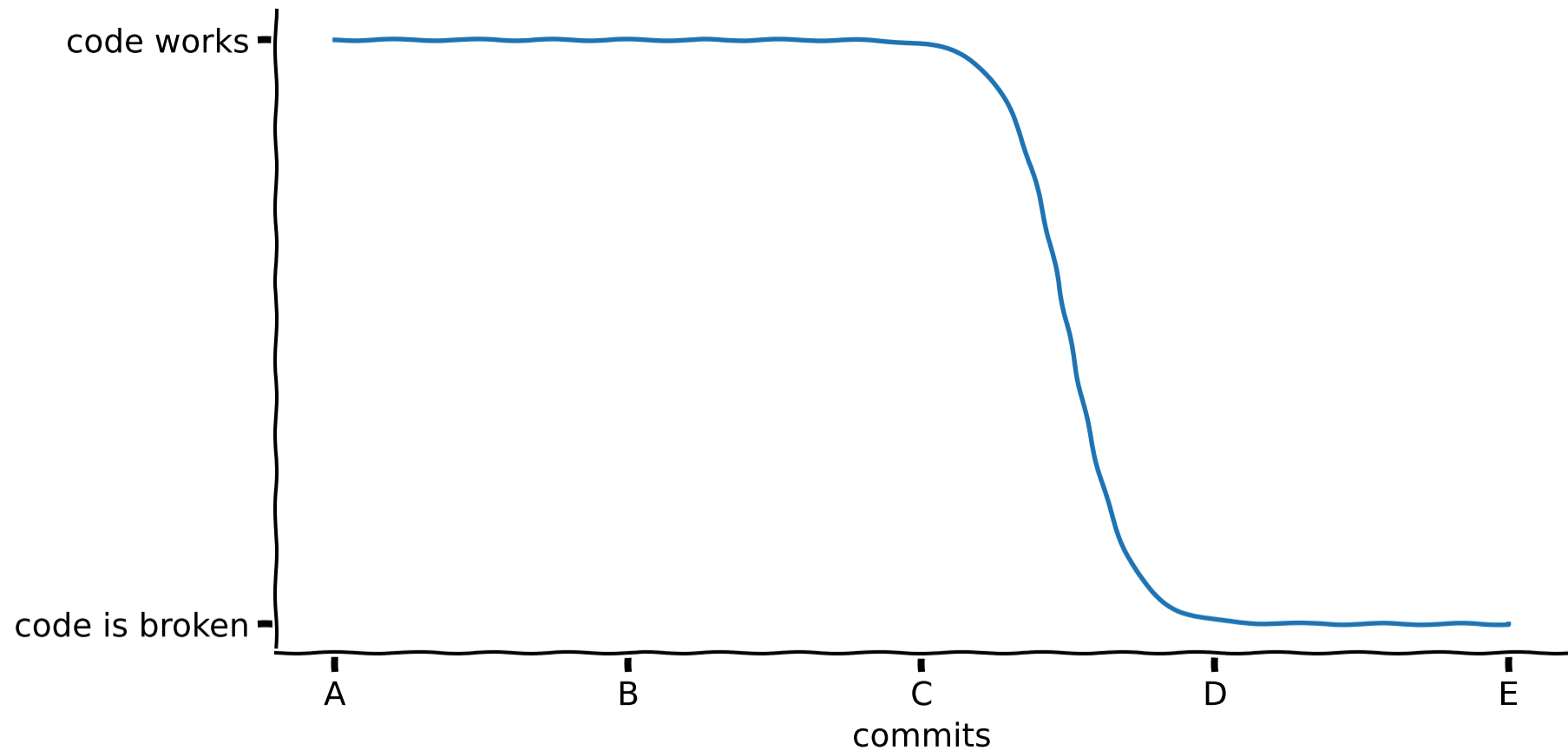
# Hands-on Session!

- Implement a bisection algorithm
- Use it to find the root of  $f$  (i.e.  $x$  such that  $f(x) == 0$ )
- How often did you have to evaluate  $f$  to obtain a result?
- How does this depend on desired accuracy?



# git bisect

“didn’t this work before?”



```
1 git bisect start E A
```



# Takeaway



# different algorithms for the same problem

Often there are different algorithms solving the same problem. These can come with **very** different complexity ratings.

# Big- $\mathcal{O}$

Helps talking about complexity, informs about large problems.



# scalar factors can matter

For small problems, scalar factors may be more important than Big- $\mathcal{O}$ .

(but the problem is small anyways)



# math can be better

*never use algorithms if math can do*

... and prefer better algorithms if math can't do



# human time matters



# Complexity Classes



$O(1)$ 

*very nice*

- basic arithmetic
- array element access
- linked list insertion or removal



$$\mathcal{O}(\log n)$$

*ok for single element operations*

- tree / dict access, insertion or deletion
- nearest neighbor search
- bisect



$$\mathcal{O}(n)$$

*nice if you have to touch many things*

- maps, reductions etc... (e.g. sum of an array)
- linked list element access
- array search



$$\mathcal{O}(n \log n)$$

*ok if you have to touch many things*

- sorting
- k-d tree construction



$$\mathcal{O}(n^2)$$

*not so nice*

- matrix multiplication
- many algorithms by accident



# what would you expect?

Think about what complexity class is expected.  
If your program behaves differently, **change it or seek advice**  
from colleagues.

# Complexity in Software Design

*“Complexity kills. It sucks the life out of developers, it makes products difficult to plan, build, and test”*

Ray Ozzie, former Microsoft CTO



# Interdependence of Components

Here complexity refers to

- amount of interactions between objects
- resulting amount of coordination and communication needed

Complexity is not about complicated designs (although simple designs are usually a good idea)



# Typical Ways to Counteract

- Use abstraction when suitable
- Encapsulate different concepts in different objects
- Refactor existing code when necessary



# Maintainability

**If code works, why put more effort in it to reduce complexity and make it more approachable?**

- Your future self and others can understand the code faster,
- Changing and adding components should be easy and only require local adjustments,
- Bugs are easier to find when the complexity is low (and components individually testable),
- This also makes testing easier and more likely to test all code paths (high code coverage)



# Resources

## Computational Complexity

- “Introduction to Algorithms” by *T. Cormen, C. Leiserson, R. Rivest* and *C. Stein*, MIT Press

## Complexity in Software Design

- “Clean Code” by *Robert Martin* [BIS-Erdsystem](#)