122COM: Profiling and Complexity

David Croft

Coventry University

david.croft@coventry.ac.uk

2016

Overview

1 Profiling
- Efficiency
- Optimization
- Profilers

2 *O*() notation
- Simple algorithms
- Good algorithms
- Bad algorithms

3 Recap

Efficiency

When writing software think about it's efficiency.

- Time.
- Memory.

Efficiency

C

When writing software think about it's efficiency.

- Time.
- Memory.
- Time vs Memory.
  - Can you trade one for the other
  - I.e. data stored in RAM costs memory but saves time.
  - I.e. data stored on hard drive saves memory but costs time.

Efficiency

When writing software think about it's efficiency.

- Time.
- Memory.
- Time vs Memory.
  - Can you trade one for the other
  - I.e. data stored in RAM costs memory but saves time.
  - I.e. data stored on hard drive saves memory but costs time.
- Optimization makes software run faster/leaner/better.

Optimization

"Premature optimization is the root of all evil"

–Knuth

Optimization

C

"Premature optimization is the root of all evil"

–Knuth

For any large piece of code you should:

Optimization

C

"Premature optimization is the root of all evil"

–Knuth

For any large piece of code you should:

- Write clear, easily understood code. Focus on getting the behaviour right, not on performance.

Optimization

"Premature optimization is the root of all evil"

–Knuth

For any large piece of code you should:

- Write clear, easily understood code. Focus on getting the behaviour right, not on performance.
- Test the performance.
  - It may be fine.

Optimization

C

"Premature optimization is the root of all evil"

–Knuth

For any large piece of code you should:

- Write clear, easily understood code. Focus on getting the behaviour right, not on performance.
- Test the performance.
  - It may be fine.
- Profile your code to get the baseline performance.
  - So that you know if you are making things better or worse.

Optimization

C

"Premature optimization is the root of all evil"

–Knuth

For any large piece of code you should:

- Write clear, easily understood code. Focus on getting the behaviour right, not on performance.
- Test the performance.
  - It may be fine.
- Profile your code to get the baseline performance.
  - So that you know if you are making things better or worse.
- Focus your efforts on the code that is consuming all the time.
  - E.g. small pieces of code that get called multiple times.

Profiler types

C

Profiling is a method of analysing your code to identify the impact of the different functions/classes/sections etc.

Instrumentation profilers

- Add extra bits of code to track time/memory/function calls.
  - Can be done manually.
  - But automatic is better.
- Accurate.
  - But slows things down.

Statistical profilers

- Regularly checks the software state.
- Accurate-ish.
  - Based on statistical sampling.
  - Doesn't slow things down.

Example

In this example which function takes the most time?

- `fast_math_function()` or `slow_math_function()`?

```python
def fast_math_function(a, b):
    time.sleep(0.00001)
    return a + b

def slow_math_function(a, b):
    time.sleep(3)
    return a + b

def main():
    for i in range(int(1.0000)):
        slow_math_function(42, 69)

    for i in range(int(100000)):
        fast_math_function(42,69)

if __name__ == '__main__':
    sys.exit(main())
```

lec_functions.py

Example

In this example which function takes the most time?

- `fast_math_function()` or `slow_math_function()`?
- Why don't we just profile it and find out?

```
def fast_math_function(a, b):
    time.sleep(0.00001)
    return a + b

def slow_math_function(a, b):
    time.sleep(3)
    return a + b

def main():
    for i in range(int(1.0000)):
        slow_math_function(42, 69)

    for i in range(int(100000)):
        fast_math_function(42,69)

if __name__ == '__main__':
    sys.exit(main())
```

lec_functions.py

Profiler results

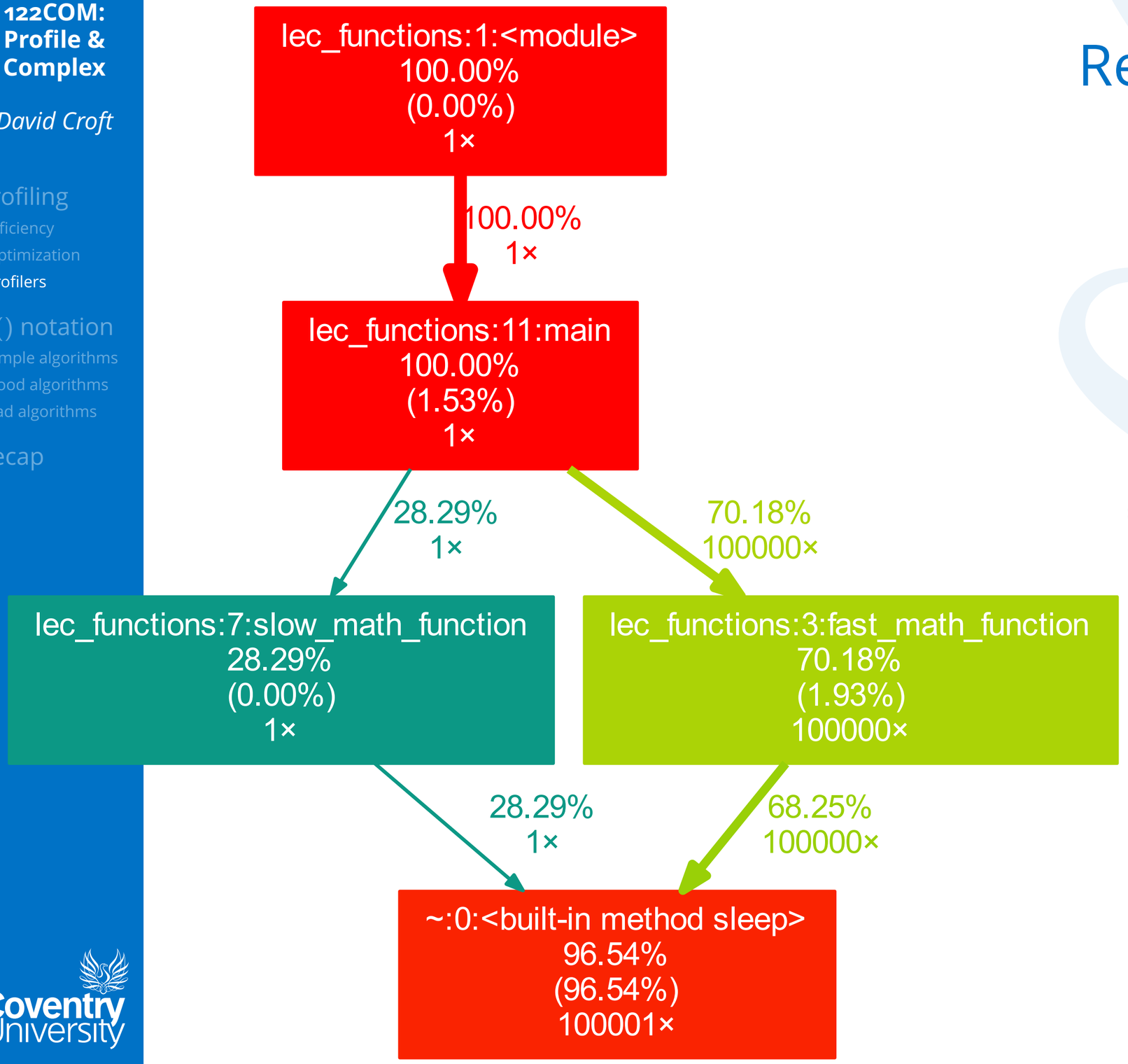

```
>> python3 -m cProfile lec_functions.py

         200007 function calls in 10.362 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000   10.362   10.362 lec_functions.py:1(<module>)
     1    0.137    0.137   10.362   10.362 lec_functions.py:11(main)
100000    0.171    0.000    7.222    0.000 lec_functions.py:3(fast_math_function)
     1    0.000    0.000    3.003    3.003 lec_functions.py:7(slow_math_function)
     1    0.000    0.000   10.362   10.362 {built-in method exec}
     1    0.000    0.000    0.000    0.000 {built-in method exit}
100001   10.054    0.000   10.054    0.000 {built-in method sleep}
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' obje
```

Things to note:

- Total time - time spent in each function.
- Cumulative time - time spent in each function AND the functions it calls.

Results visualised

C



Results passed through Graphviz/gprof2dot.

- A profiling visualisation tool.

O() notation

C

Profiling is very useful in determining the actual performance of your code.

- Unexpected bottlenecks.
- Problems in 3rd party libraries etc.

- Not so good at measuring how code will scale.
  - Change in response to different inputs.
- Algorithmic complexity.
- Certain algorithms are known to be better than other algorithms.

O() notation

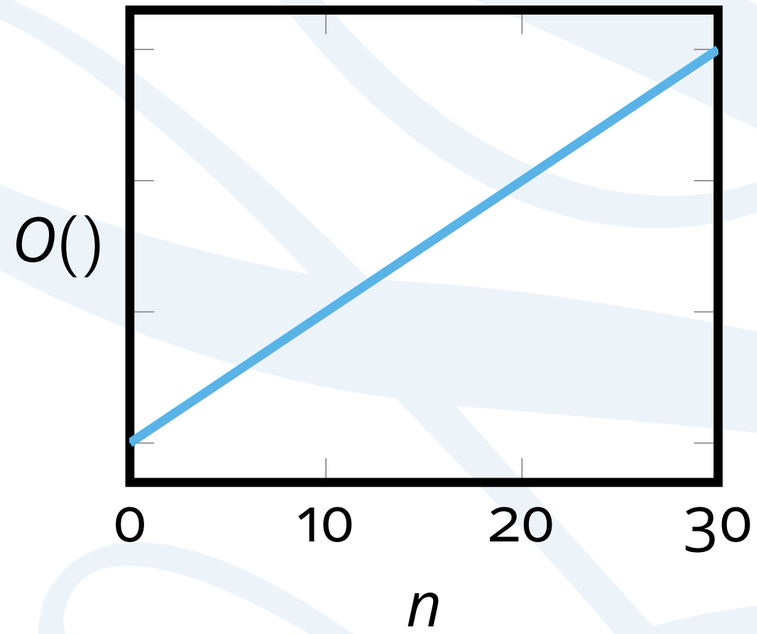
Used to describe complexity in terms of time and/or space.

- Commonly encountered examples...
  - $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(2^n)$ and $O(n!)$
- n refers to the size of the problem.
  - E.g. n values to be sorted.
  - E.g. n values to be searched.
- *O*() notation describes the worst case scenario.
  - Usually, unless otherwise stated.
- *O*() notation is discussed in detail next year.
  - Main idea is to capture the dominant term: the thing that is most important when the size of the input (n) gets big.

$O(n)$ C

Linear complexity.

- *n* is directly proportional to time/space required
  - E.g. *n* doubles then time/space doubles.
- E.g. linear/sequential search.



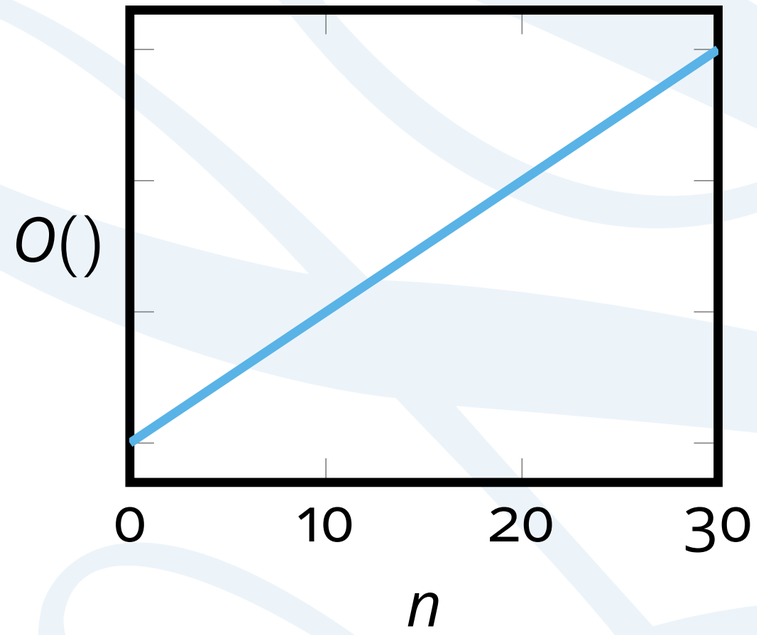

```
a = [ 0, 1, 2, 3, 4, 5, 6, 7, 42 ]

for i in a:
  if i == 42:                    (n)
    print('Found it')            (1)
    break                        (1)
```

- So the algorithm takes $n + n + 1 + 1 = 2n + 2$ operations.
  - BUT! We would say it has complexity $O(n)$ as when *n* gets big the factor or 2 and addition of 2 become irrelevant.

$O(n)$ C

Linear complexity.

- *n* is directly proportional to time/space required
  - E.g. *n* doubles then time/space doubles.
- E.g. linear/sequential search.




```
a = [ 0, 1, 2, 3, 4, 5, 6, 7, 42 ]

for i in a:                          (n)
  if i == 42:                        (n)
    print('Found it')                (1)
    break                            (1)
```

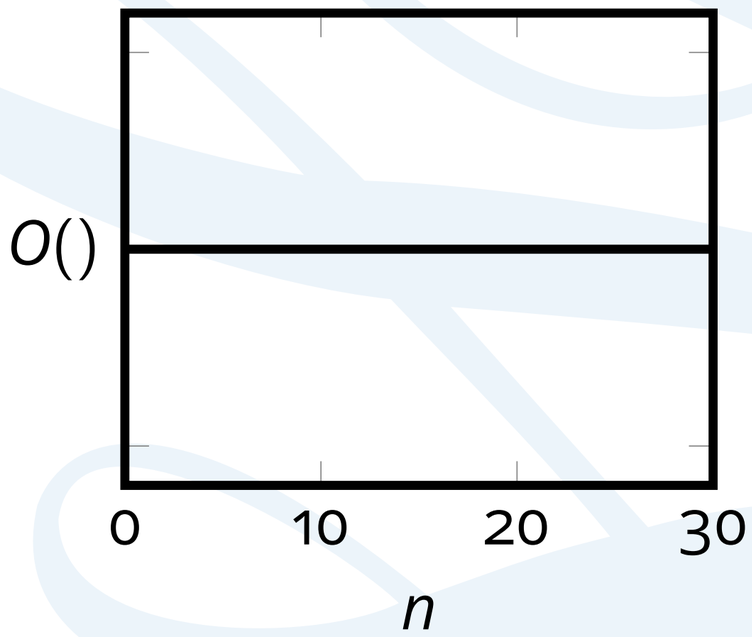
- So the algorithm takes $n + n + 1 + 1 = 2n + 2$ operations.
  - BUT! We would say it has complexity $O(n)$ as when *n* gets big the factor or 2 and addition of 2 become irrelevant.

$O(1)$ C

Constant complexity.

- *n* doesn't matter.
- Always takes same time/space.
- E.g. getting first item in an array.

```
a = [ i for i in range(100) ]
b = [ i for i in range(1000000) ]

print(a[0])
print(b[0])
```



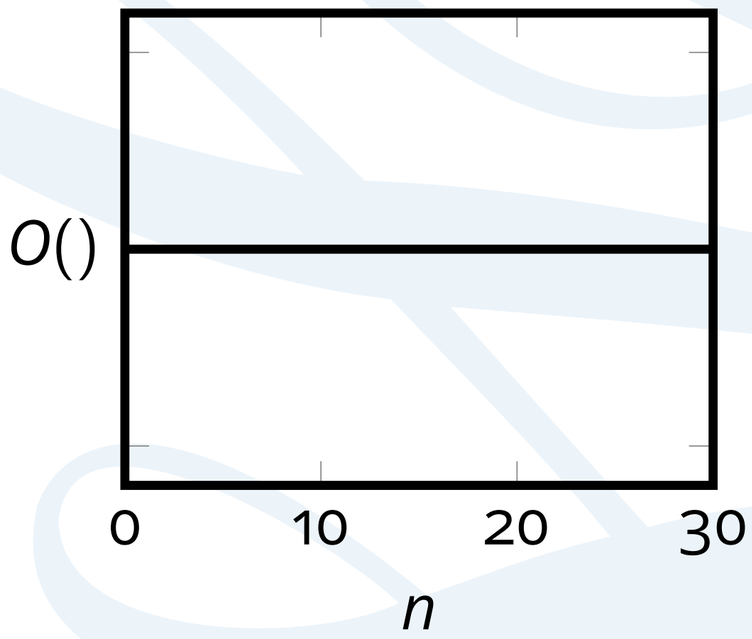
$O(1)$ C

Constant complexity.

- *n* doesn't matter.
- Always takes same time/space.
- E.g. getting first item in an array.

```
a = [ i for i in range(100) ]
b = [ i for i in range(1000000) ]

print(a[0])
print(b[0])
```

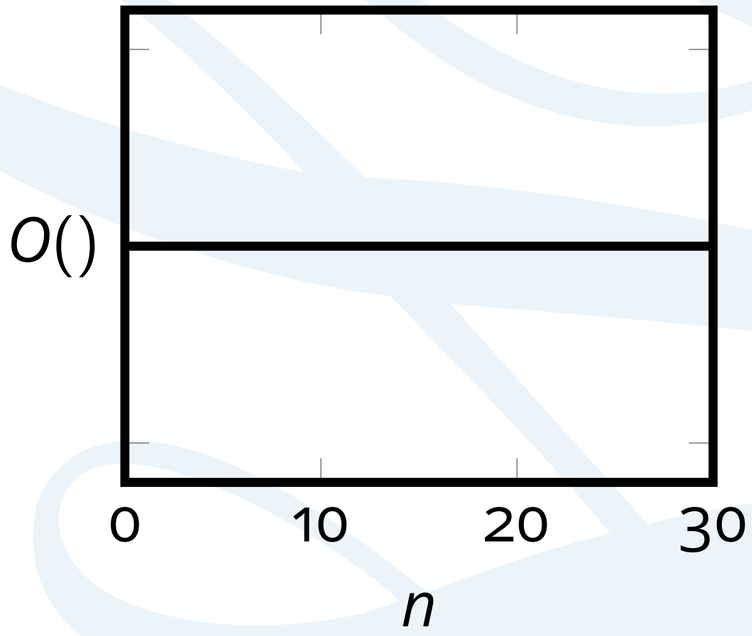
(1)



$O(1)$ C

Constant complexity.

- *n* doesn't matter.
- Always takes same time/space.
- E.g. getting first item in an array.



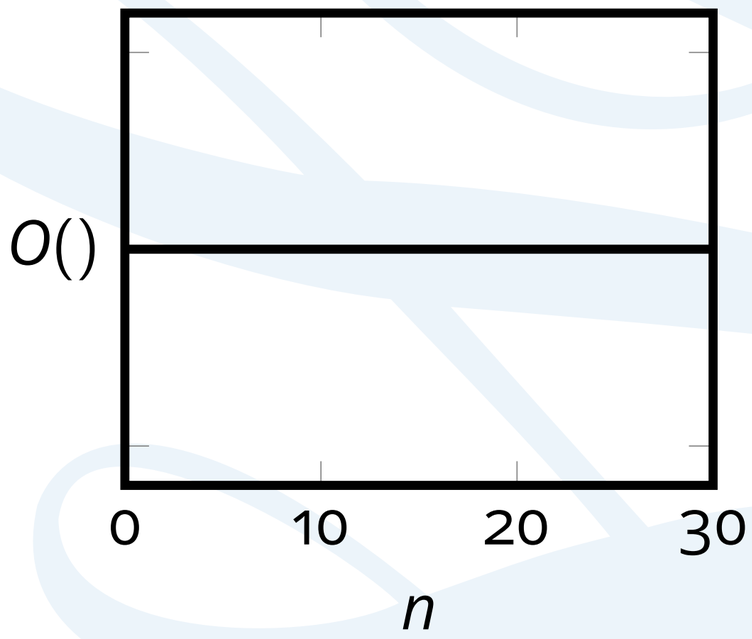

```
a = [ i for i in range(100) ]
b = [ i for i in range(1000000) ]

print(a[0])                    (1)
print(b[0])                    (1)
```

$O(1)$ C

Constant complexity.

- *n* doesn't matter.
- Always takes same time/space.
- E.g. getting first item in an array.

```
a = [ i for i in range(100) ]          (n)
b = [ i for i in range(1000000) ]      (m)

print(a[0])                            (1)
print(b[0])                            (1)
```

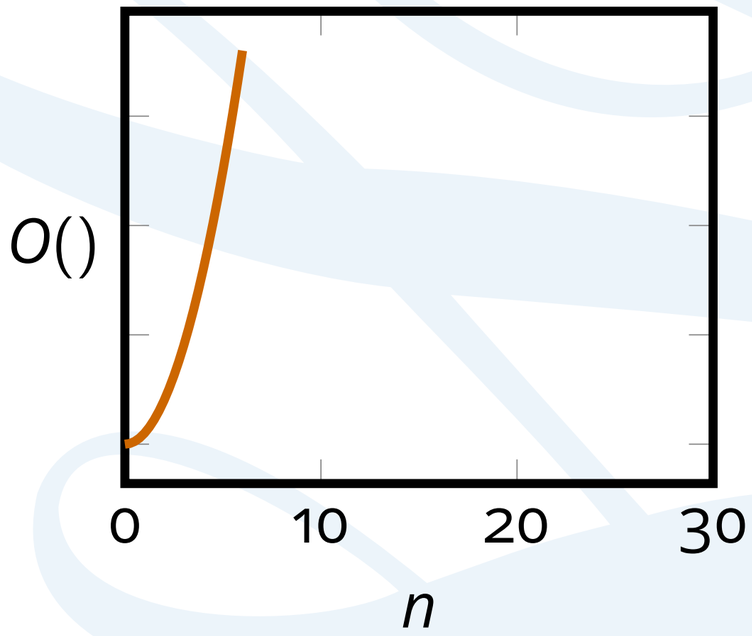


$O(n^2)$ I

Quadratic complexity.

- A lot of simple sorting algorithms are $O(n^2)$.
- Nested `for` loops are common example.
- $O(n^3)$, $O(n^4)$, $O(n^m)$ etc. are all possible.
- Polynomial time.

```
print('The n times tables')

for i in range(n):
  for j in range(n):
    print(i*j)
```

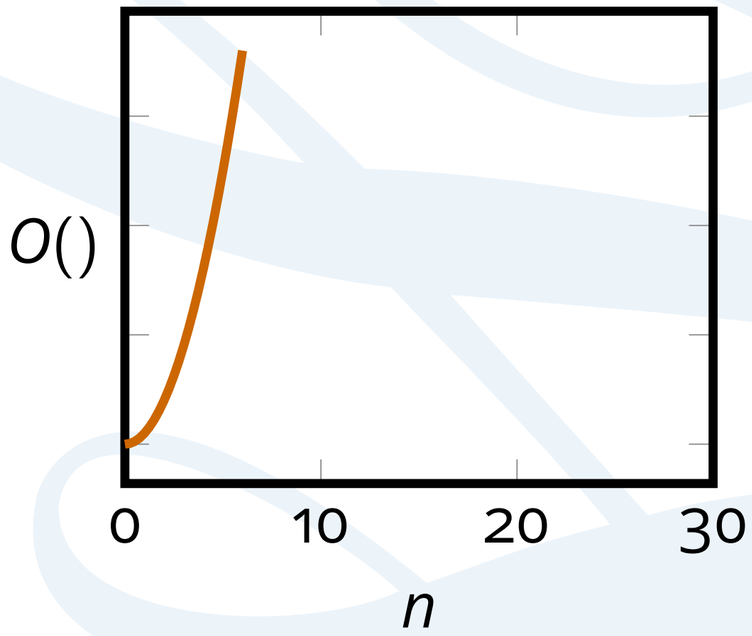


$O(n^2)$ I

Quadratic complexity.

- A lot of simple sorting algorithms are $O(n^2)$.
- Nested `for` loops are common example.
- $O(n^3)$, $O(n^4)$, $O(n^m)$ etc. are all possible.
- Polynomial time.

```
print('The n times tables')  (1)

for i in range(n):
  for j in range(n):
    print(i*j)
```



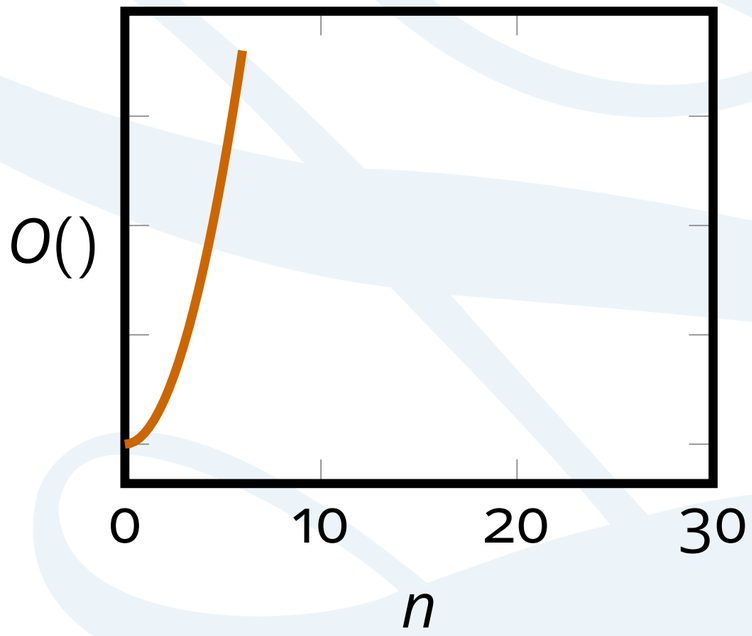
$O(n^2)$

I

Quadratic complexity.

- A lot of simple sorting algorithms are $O(n^2)$.
- Nested `for` loops are common example.
- $O(n^3)$, $O(n^4)$, $O(n^m)$ etc. are all possible.
- Polynomial time.

```
print('The n times tables')   (1)

for i in range(n):            (n)
  for j in range(n):
    print(i*j)
```

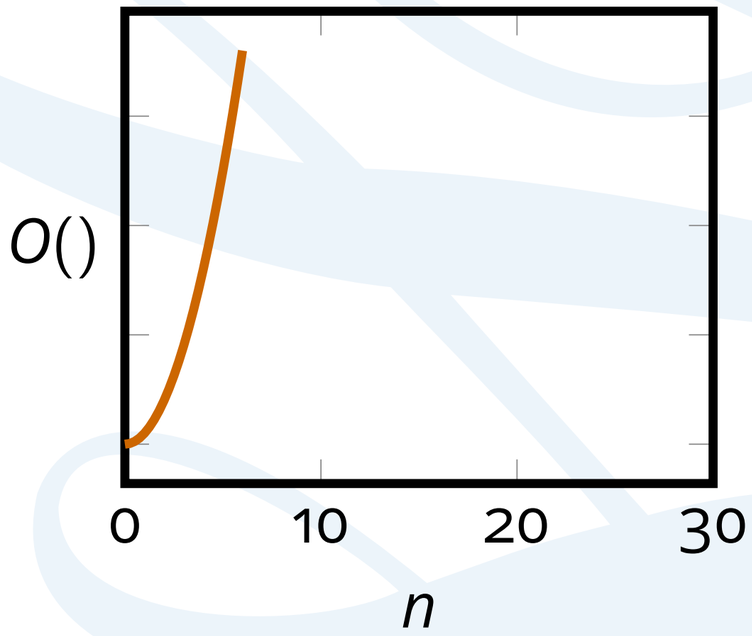


$O(n^2)$

I

Quadratic complexity.

- A lot of simple sorting algorithms are $O(n^2)$.
- Nested `for` loops are common example.
- $O(n^3)$, $O(n^4)$, $O(n^m)$ etc. are all possible.
- Polynomial time.

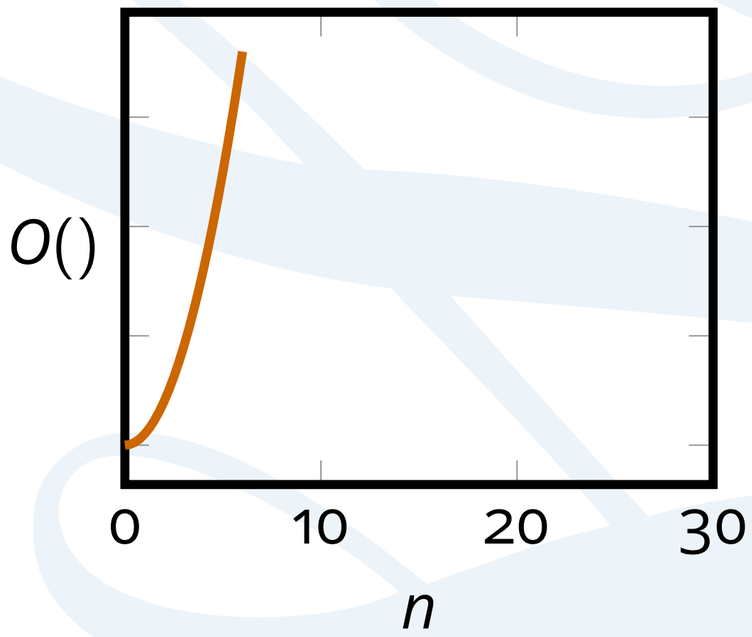



```
print('The n times tables')   (1)

for i in range(n):            (n)
  for j in range(n):          (n*n)
    print(i*j)
```

$O(n^2)$ I

Quadratic complexity.

- A lot of simple sorting algorithms are $O(n^2)$.
- Nested `for` loops are common example.
- $O(n^3)$, $O(n^4)$, $O(n^m)$ etc. are all possible.
- Polynomial time.



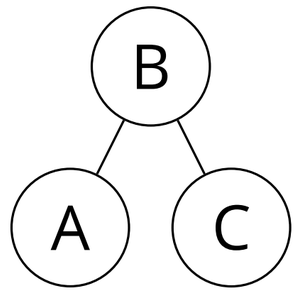

```
print('The n times tables')   (1)

for i in range(n):            (n)
  for j in range(n):          (n*n)
    print(i*j)                (n*n)
```

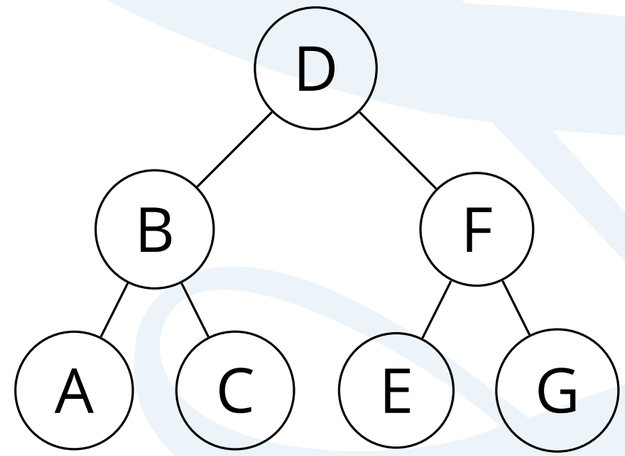
$O(\log n)$

I

Logarithmic complexity.

- Bit more complicated.
- Binary search.



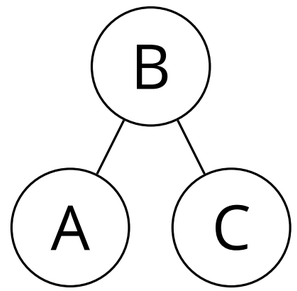


$O(\log n)$

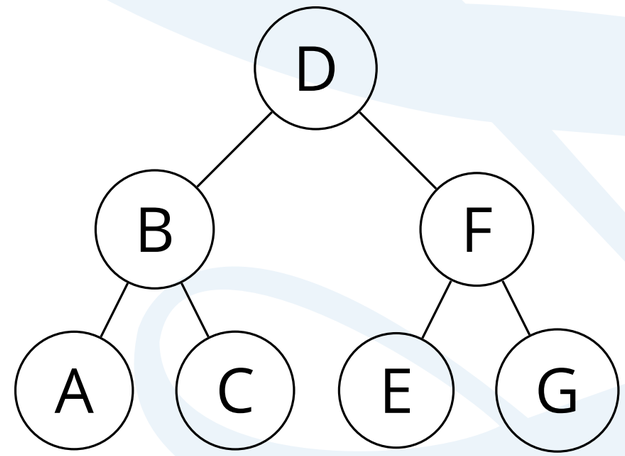
I

Logarithmic complexity.

- Bit more complicated.
- Binary search.



$n = 3$

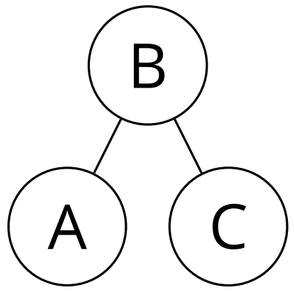


$O(\log n)$

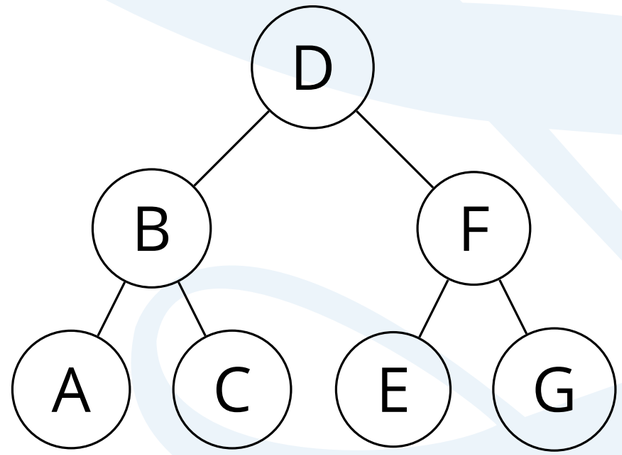
I

Logarithmic complexity.

- Bit more complicated.
- Binary search.



$$n = 3$$
$$O(\log n) = 1.58 \Rightarrow 1$$

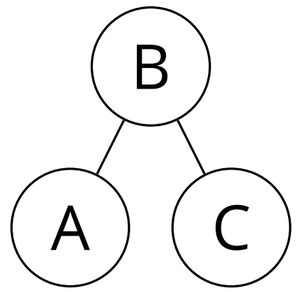


$O(\log n)$

I

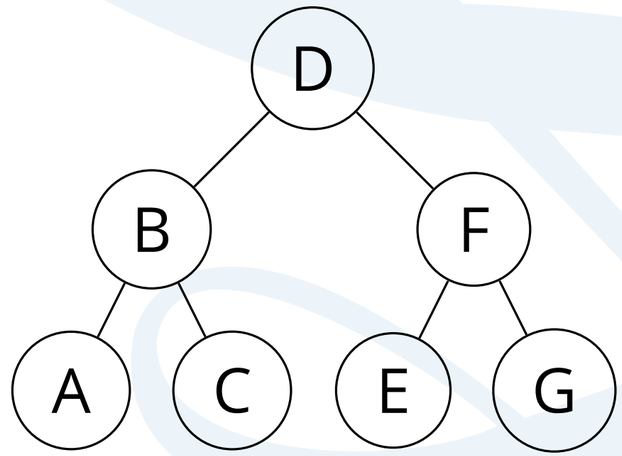
Logarithmic complexity.

- Bit more complicated.
- Binary search.



$n = 3$

$O(\log n) = 1.58 \Rightarrow 1$



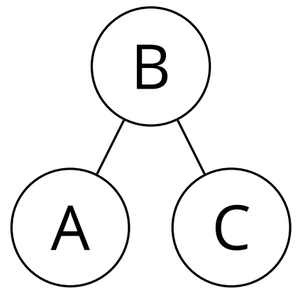
$n = 7$

$O(\log n)$

I

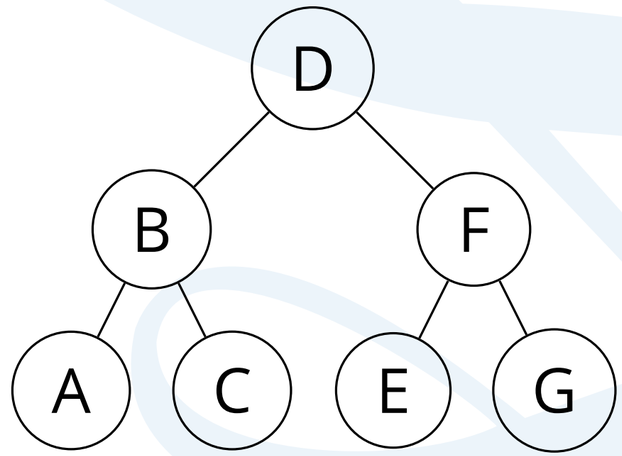
Logarithmic complexity.

- Bit more complicated.
- Binary search.



$n = 3$

$O(\log n) = 1.58 \Rightarrow 1$



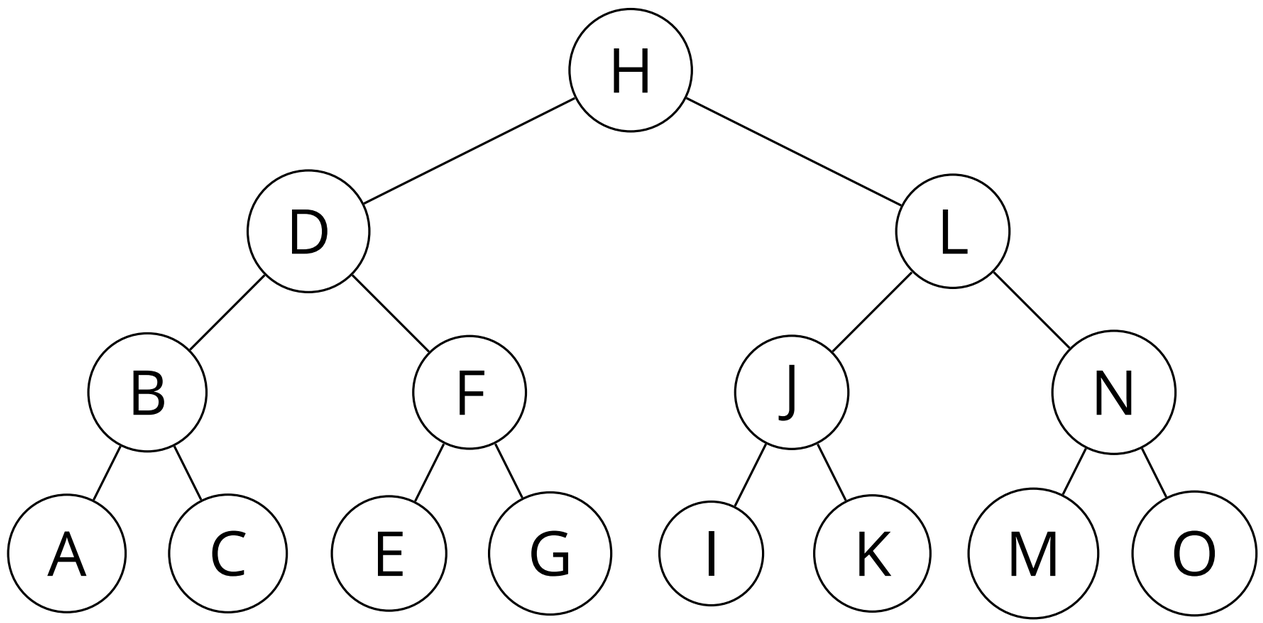
$n = 7$

$O() = 2.81 \Rightarrow 2$

$O(\log n)$ cont.

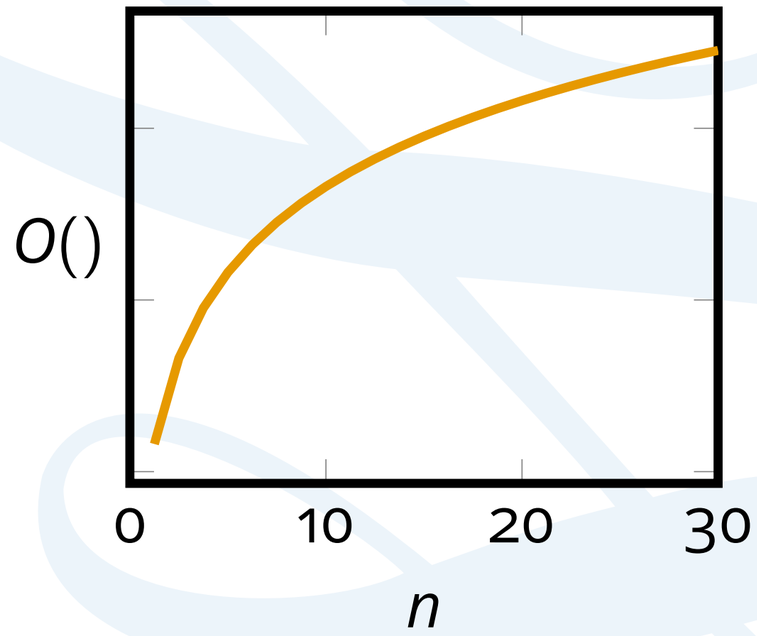
$O(\log n)$ complexity.

- Increases very slowly.
- $\log_2(100)$ is only 6.
- $\log_2(1000000000000)$ (trillion) is only 39.



$n = 15$

$O() = 3.91 \Rightarrow 3$

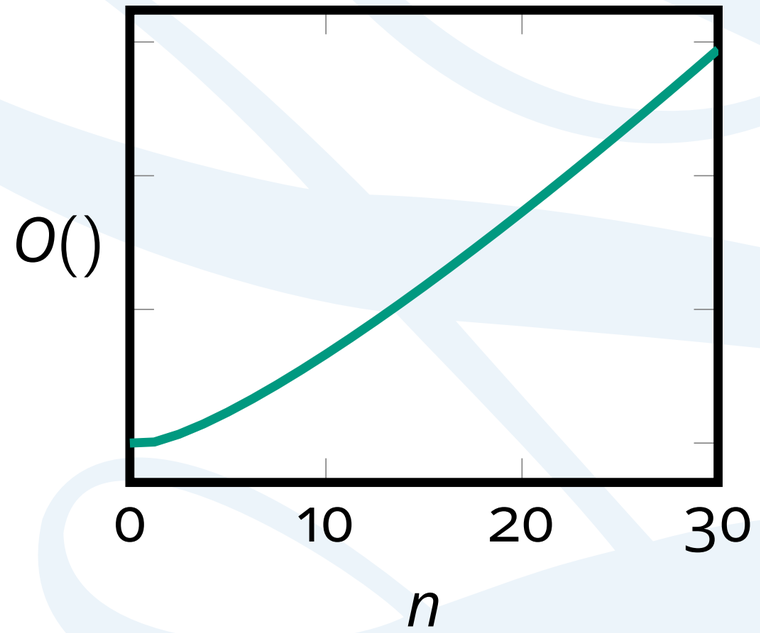


$O(n \log n)$

A

Loglinear complexity.

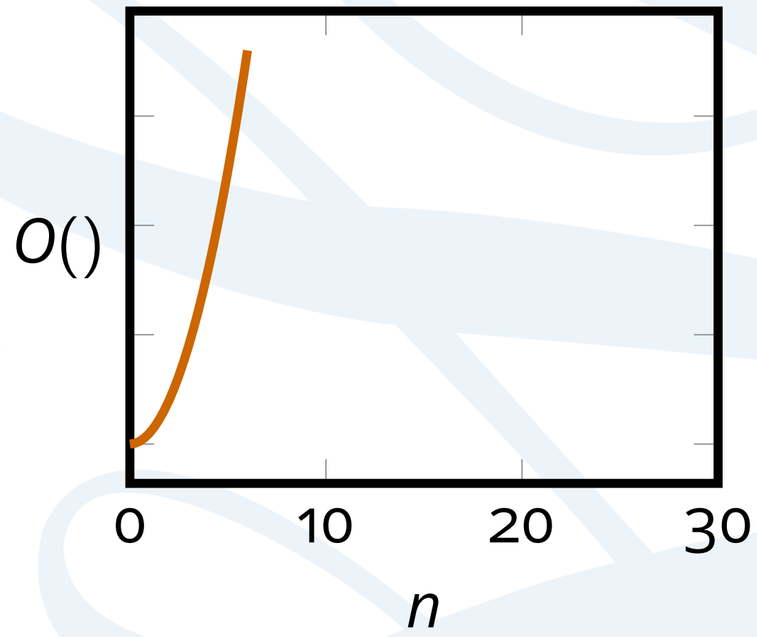
- Looks more difficult than it is.
- $O(n \log n)$ means, do $O(\log n)$ n times.
- E.g. binary search for n items.
  - Binary search is $O(\log n)$.
  - Doing n binary searches.
  - So $O(n \log n)$.
- Lots of good sorting algorithms are $O(n \log n)$.



$O(2^n)$ A

Exponential complexity.

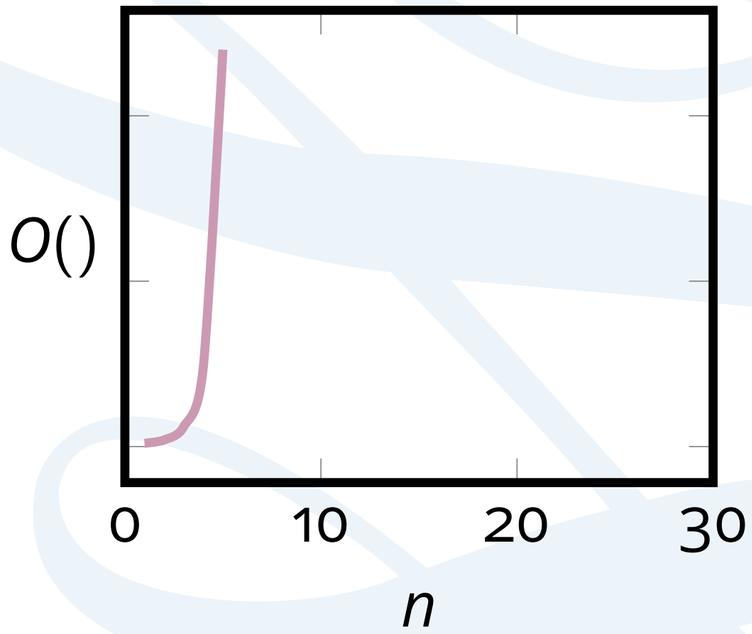
- Very, very bad.
- Each additional value doubles the time/space.
- Doesn't scale.
- $O(3^n)$, $O(4^n)$ etc. are all possible.



$O(n!)$ A

Factorial complexity.

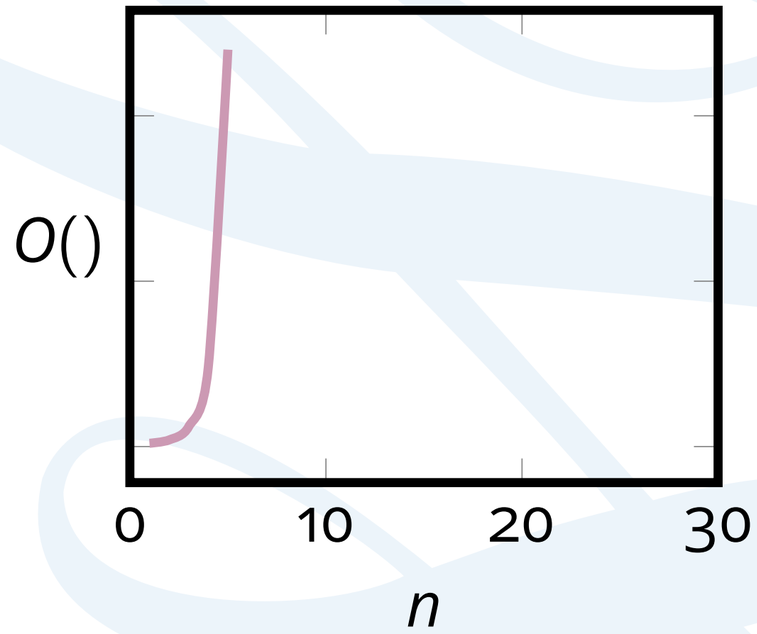
- Just awful.
- Every possible combination of n items.
- Brute force travelling salesman is $O(n!)$.
- Totally impractical even for small values of n.

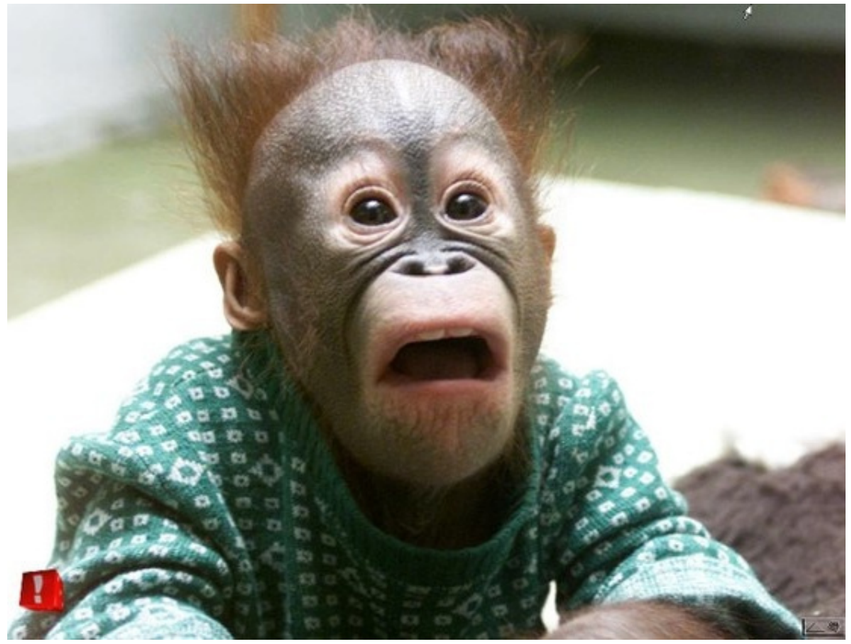


$O(n!)$ A

Factorial complexity.

- Just awful.
- Every possible combination of n items.
- Brute force travelling salesman is $O(n!)$.
- Totally impractical even for small values of n.





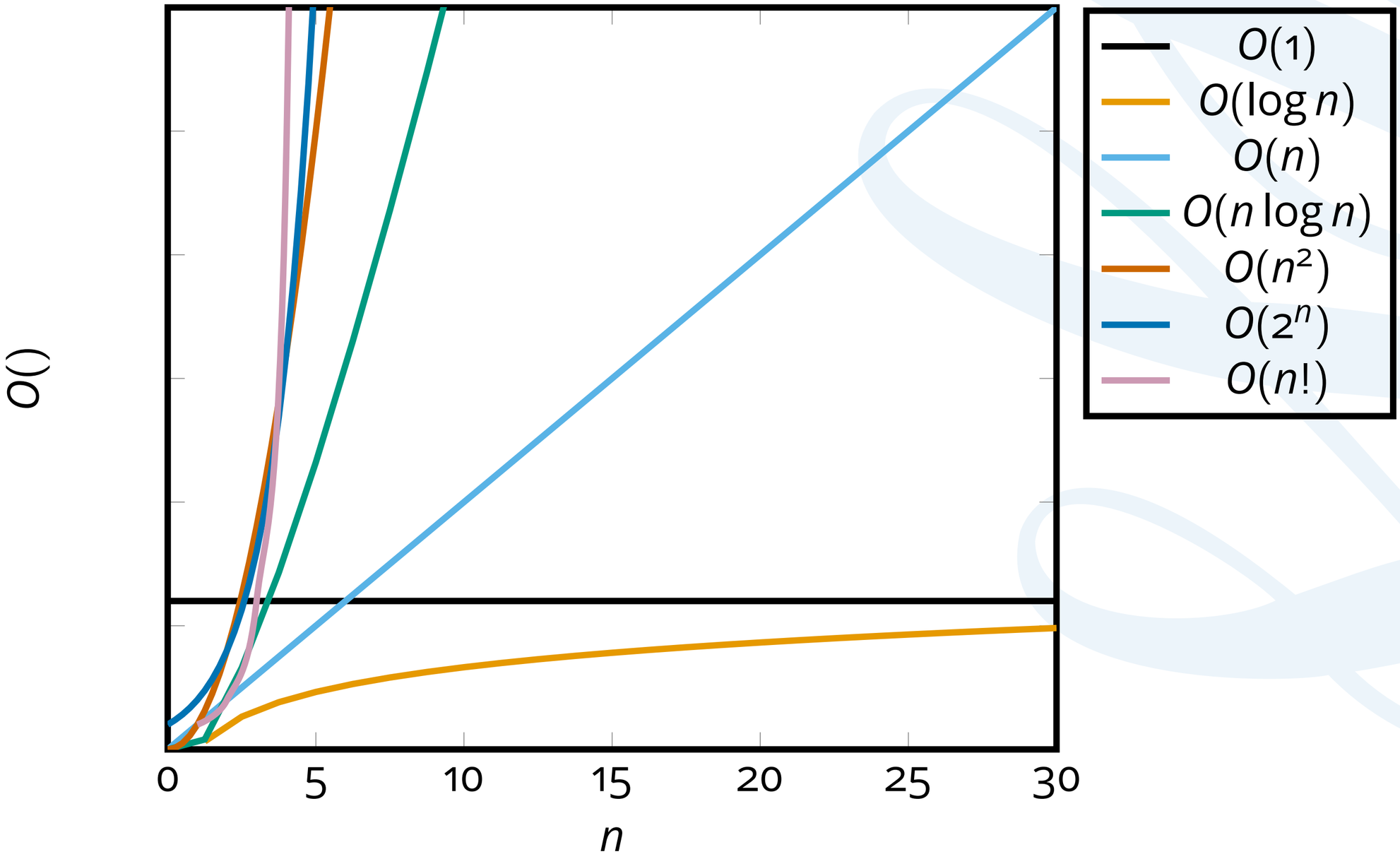
Scaling

Different *O*() == wildly different complexity.

		n = 2	10	100
Best	$O(1)$	1	1	1
	$O(\log n)$	1	3	6
↑	$O(n)$	2	10	100
↓	$O(n \log n)$	2	33	664
	$O(n^2)$	4	100	10000
	$O(2^n)$	4	1024	$1.27 \cdot 10^{30}$
Worst	$O(n!)$	2	3628800	$9.33 \cdot 10^{157}$

Comparison

A



Complexity vs. Time

Complexity isn't the same as efficiency.

- A good $O(n^2)$ implementation can be better than a bad $O(n)$.
  - For a while.
- Eventually, as n increases, $O(n)$ will always outperform $O(n^2)$ etc.

Complexity vs. Time

Complexity isn't the same as efficiency.

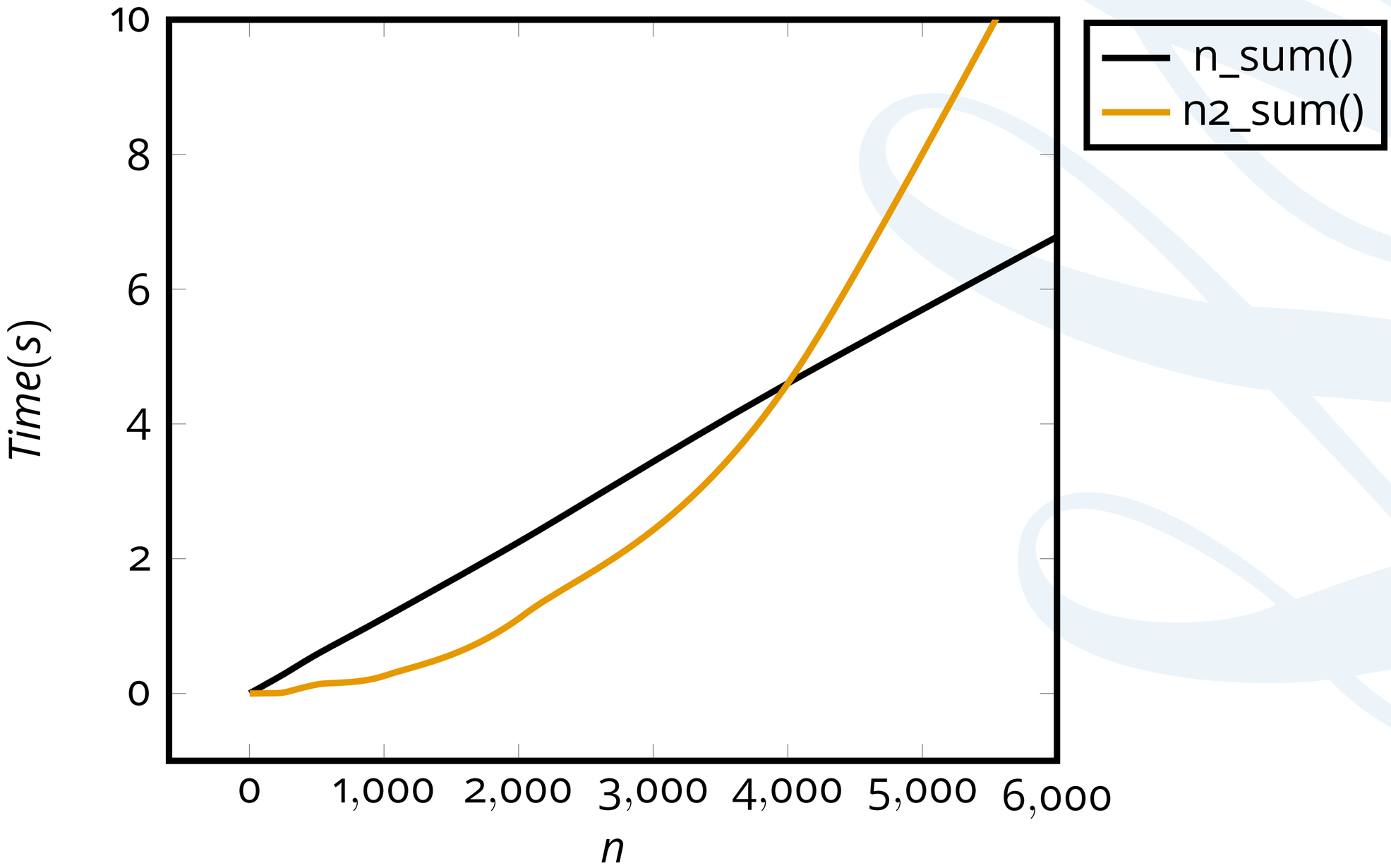
- A good $O(n^2)$ implementation can be better than a bad $O(n)$.
  - For a while.
- Eventually, as n increases, $O(n)$ will always outperform $O(n^2)$ etc.

```python
def n_sum(sequence):
    total = 0
    for i in range(len(sequence)):
        total += sequence[i]
        time.sleep(0.001)
    return total
```

lec_fast_slow_functions.py

```python
def n2_sum(sequence):
    total = 0
    for i in range(len(sequence)):
        counter = 0
        while counter < i:
            counter += 1
        total += sequence[counter]

    return total
```

Time results



Quiz

Recap

Profiling help determines the actual performance of your code.

- Statistical profilers.
  - Accurate-ish
- Instrumental profilers.
  - Insert additional instructions.
  - Accurate but slows things down.

O() describes algorithm complexity.

- Time/space.
- How your code should scale.
  - L

  ots of real world issues can mess it up.
  - Memory limits etc.
- $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!)$
- $\geq O(2^n)$ means exponential.
- $< O(n^2)$ means polynomial.

The End