Introduction
Fibonacci example
Difficulty
Module content
Profiling
Efficiency
Optimization
$O()$ notation
Simple algorithms
Good algorithms
Bad algorithms
Recap

122COM: Introduction to algorithms

David Croft

Coventry University

david.croft@coventry.ac.uk

2017

Overview

1 Introduction

2 Fibonacci example

3 Difficulty

4 Module content

5 Profiling
- Efficiency
- Optimization

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

7 Recap

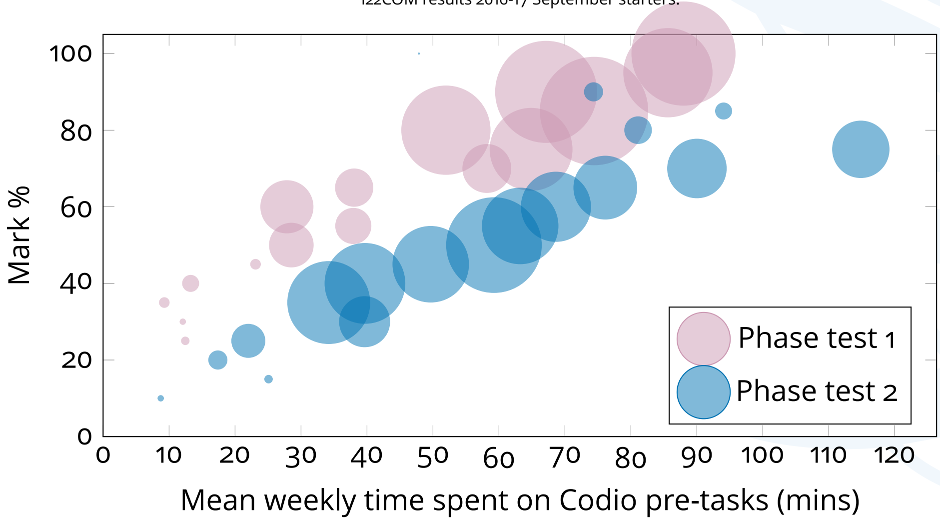
Introduction
Fibonacci example
Difficulty
Module content
Profiling
Efficiency
Optimization
O() notation
Simple algorithms
Good algorithms
Bad algorithms
Recap

Expectations

C

You have all attempted the green Codio exercises for this week.

122COM results 2016-17 September starters.



Introduction
Fibonacci example
Difficulty
Module content
Profiling
Efficiency
Optimization
$O()$ notation
Simple algorithms
Good algorithms
Bad algorithms
Recap

Introduction

C

Introduction to algorithms module.

- What is an algorithm?

Introduction

C

Introduction to algorithms module.

- What is an algorithm?
- Not the same as code.
- Not the same as a program.

Introduction

Fibonacci example

Difficulty

Module content

Profiling

Efficiency

Optimization

$O()$ notation

Simple algorithms

Good algorithms

Bad algorithms

Recap

Introduction

C

Introduction to algorithms module.

- What is an algorithm?
- Not the same as code.
- Not the same as a program.

Task/algorithm/code

C

A task is a problem that needs to be solved.

- I.e. bake me a cake.

Introduction
Fibonacci example
Difficulty
Module content
Profiling
Efficiency
Optimization
O() notation
Simple algorithms
Good algorithms
Bad algorithms
Recap

Task/algorithm/code

C

A task is a problem that needs to be solved.

- I.e. bake me a cake.

An algorithm is a generalised set of instructions to perform a specific task.

- A strategy to solve a given problem.
  - Many different strategies to solve same task.
- Like a recipe.

Task/algorithm/code

C

A task is a problem that needs to be solved.

- I.e. bake me a cake.

An algorithm is a generalised set of instructions to perform a specific task.

- A strategy to solve a given problem.
  - Many different strategies to solve same task.
- Like a recipe.

Code is a specific set of instructions to perform a specific task.

- An implementation of a strategy in a specific language/system.
- Have to adapt the algorithm to the specific features and abilities of the language.

Introduction
Fibonacci example
Difficulty
Module content
Profiling
Efficiency
Optimization
O() notation
Simple algorithms
Good algorithms
Bad algorithms
Recap

Fibonacci sequence algorithm

C

Task - calculate the fibonacci sequence.

Fibonacci sequence algorithm

C

Task - calculate the fibonacci sequence.

Algorithm

1. Starting with 0 and 1.
2. Sum the two numbers to make a third.
3. Discard the lowest number.
4. Repeat from step 2.

Fibonacci sequence algorithm

C

Task - calculate the fibonacci sequence.

Algorithm

1. Starting with 0 and 1.
2. Sum the two numbers to make a third.
3. Discard the lowest number.
4. Repeat from step 2.

Recursive Python

```python
def fibonacci( a, b ):
    c = a + b
    a, b = b, c

    print( a )
    fibonacci( a, b )

fibonacci( 0, 1 )
```

Iterative C++

```cpp
for( int a=0, b=1, c;
     a>=0;
     c=a+b, a=b, b=c )
{
    cout << a << endl;
}
```

Solvable/Unsolvable

C

Some problems we can solve perfectly.

- Easy problems.
  - Fibonacci sequence.
  - Searching algorithms.
  - Polynomial time.

Some problems we can't solve.

- Hard problems.
  - Because they are provably unsolvable.
    - Literally impossible.
    - Investigate the Halting State problem.
  - Because they take too long to solve.

Heuristics

C

Some problems we could solve perfectly if only we had infinite computers/time.

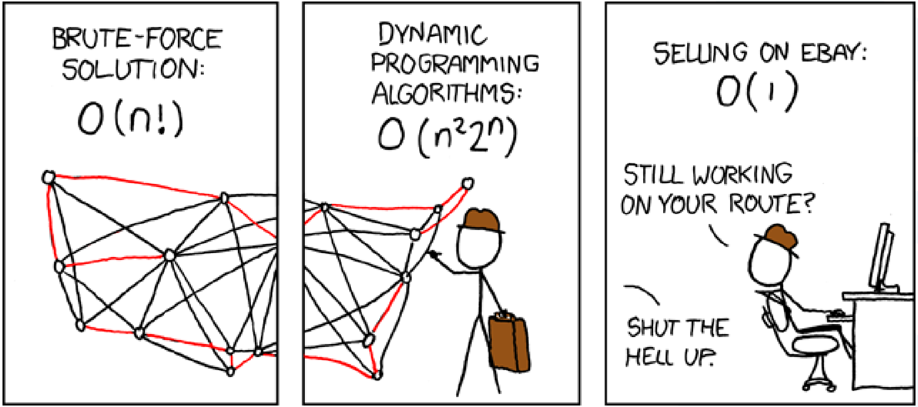
- Travelling salesman.
  - Hard problem, non-polynomial (will discuss later).
  - Can only solve very simple versions of the problem perfectly.
  - 5 cities = 120 possible solutions, 20 cities = 2 432 902 008 176 640 000 possible solutions.

Heuristic algorithms.

- Don't promise to find the best solution.
- Quickly find a 'good enough' solution.

Heuristics

A



https://xkcd.com/399/

Module content

Looking at searching and sorting algorithms in later weeks.

Will be tested on some algorithmic concepts.

- Implement simple algorithms.
- Describe advantages/disadvantages of certain algorithms.
- Big O notation.
  - How algorithms scale.
- Calculate an algorithm's $O()$ notation.

Introduction
Fibonacci example
Difficulty
Module content
Profiling
Efficiency
Optimization
$O()$ notation
Simple algorithms
Good algorithms
Bad algorithms
Recap

Efficiency

C

When writing software need to think about its efficiency.

- Time.
- Memory.

Efficiency

C

When writing software need to think about its 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.

But don't be too concerned with performance.

Introduction
Fibonacci example
Difficulty
Module content
Profiling
Efficiency
Optimization
O() notation
Simple algorithms
Good algorithms
Bad algorithms
Recap

Optimization

C

“Premature optimization is the root of all evil”

–Donald Knuth

For any large piece of code you should:

Optimization

C

"Premature optimization is the root of all evil"

–Donald Knuth

For any large piece of code you should:

- Write clear, easily understood code. Focus on getting the behaviour right, not on performance.
  - If it doesn't work then no-one cares how efficiently it fails.
- Test the performance.
  - It may be fine.
- Measure your code to get the baseline performance.
  - So that you know if you are making things better or worse.
- Ideally using profiling tools.
  - Investigate in your own time.

O() notation

C

Measuring performance identifies how your code is running for **those** inputs on **that** machine.

- Not good at measuring how code will scale.
  - Change in response to different inputs.
  - Change in response to problem size.
- Algorithmic complexity.
  - *O*() notation or Big-O notation.
- Certain algorithms are known to be better than other algorithms.

Introduction
Fibonacci example
Difficulty
Module content
Profiling
Efficiency
Optimization
$O()$ notation
Simple algorithms
Good algorithms
Bad algorithms
Recap

$O()$ 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.

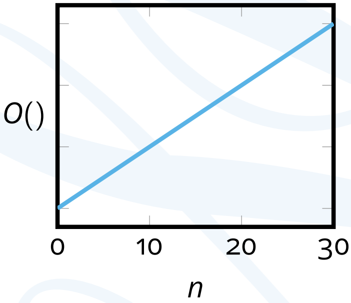
$O(n)$ C

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:
    print('Found it')
    break
```



$O(n)$

C

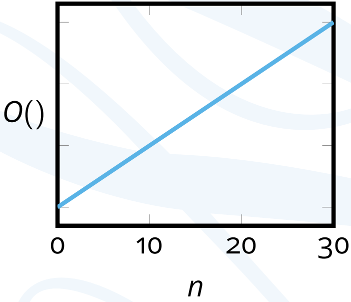
Linear complexity.

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

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

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

(n)



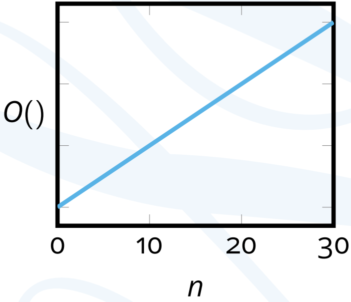
$O(n)$ C

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')
    break
```



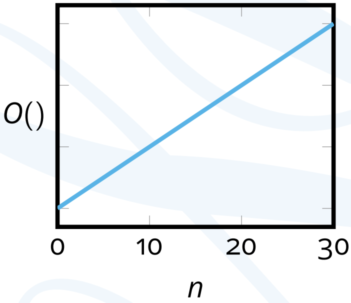
$O(n)$ C

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
```



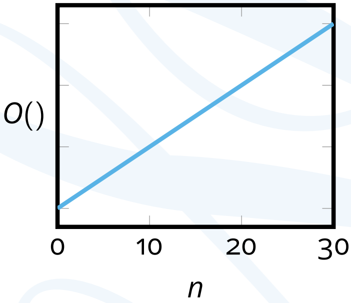
$O(n)$ C

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)
```



$O(n)$

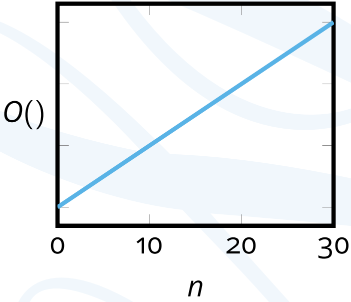
C

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 $O(n)$, constant values are irrelevant.

Introduction
Fibonacci example
Difficulty
Module content
Profiling
Efficiency
Optimization
O() notation
Simple algorithms
Good algorithms
Bad algorithms
Recap

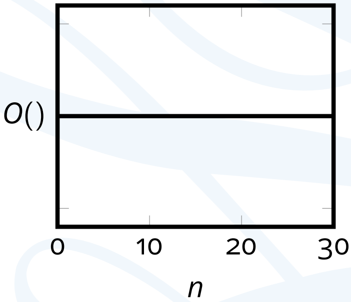
$O(1)$ C

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])
```



Introduction
Fibonacci example
Difficulty
Module content
Profiling
Efficiency
Optimization
O() notation
Simple algorithms
Good algorithms
Bad algorithms
Recap

$O(1)$

C

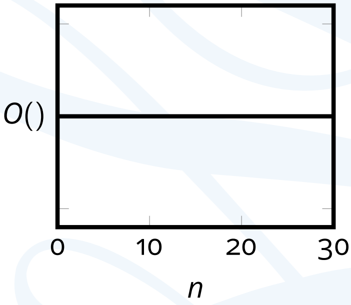
Constant complexity.

- *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)

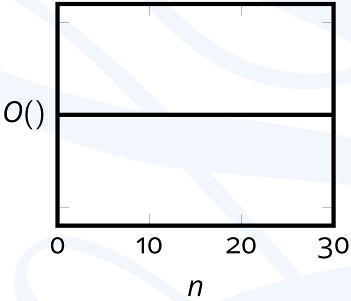
C

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)
```



Introduction
Fibonacci example
Difficulty
Module content
Profiling
Efficiency
Optimization
O() notation
Simple algorithms
Good algorithms
Bad algorithms
Recap

$O(1)$

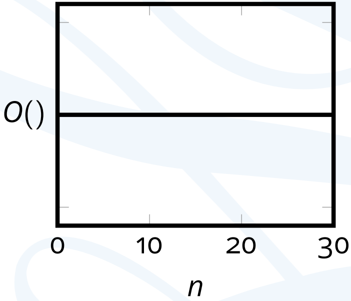
C

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)
```



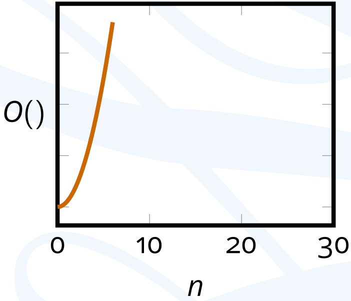
$O(n^2)$

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)
```



$O(n^2)$

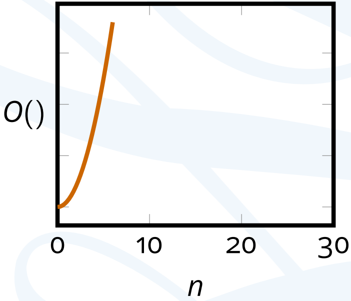
I

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)
```



$O(n^2)$

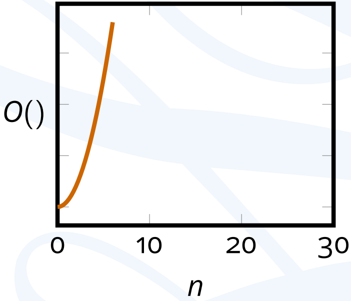
I

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)
```



Introduction
Fibonacci example
Difficulty
Module content
Profiling
Efficiency
Optimization
$O()$ notation
Simple algorithms
Good algorithms
Bad algorithms
Recap

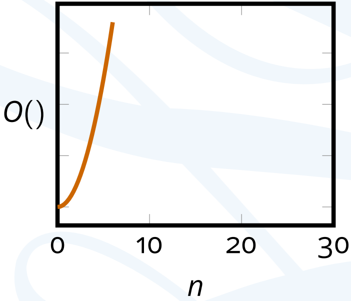
$O(n^2)$

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)
```



$O(n^2)$

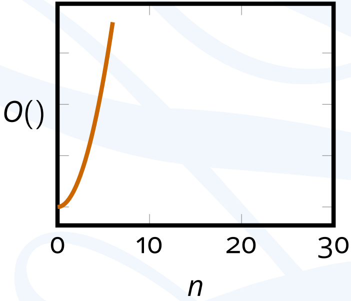
I

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)
```

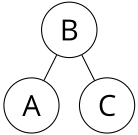


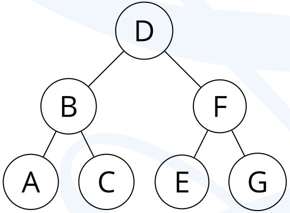
$O(\log n)$

I

Logarithmic complexity.

- Bit more complicated.
- Imagine you want to find a node in a tree (i.e. binary search).
  - How many nodes would you have to check?.



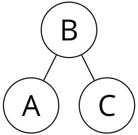


$O(\log n)$

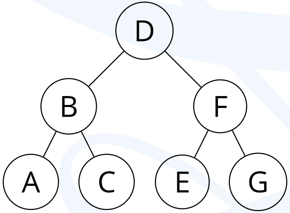
I

Logarithmic complexity.

- Bit more complicated.
- Imagine you want to find a node in a tree (i.e. binary search).
  - How many nodes would you have to check?.



$n = 3$

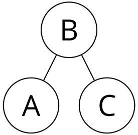


Coventry University

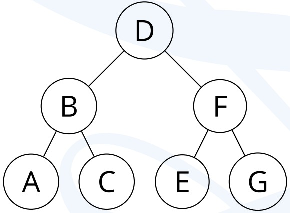
$O(\log n)$ I

Logarithmic complexity.

- Bit more complicated.
- Imagine you want to find a node in a tree (i.e. binary search).
  - How many nodes would you have to check?.



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

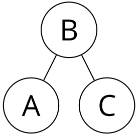


$O(\log n)$

I

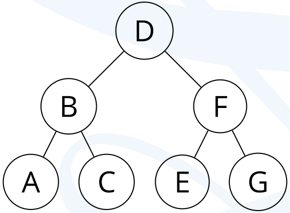
Logarithmic complexity.

- Bit more complicated.
- Imagine you want to find a node in a tree (i.e. binary search).
  - How many nodes would you have to check?.



$n = 3$

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



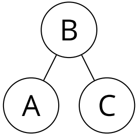
$n = 7$

$O(\log n)$

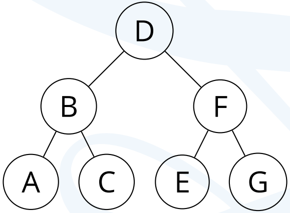
I

Logarithmic complexity.

- Bit more complicated.
- Imagine you want to find a node in a tree (i.e. binary search).
  - How many nodes would you have to check?.



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



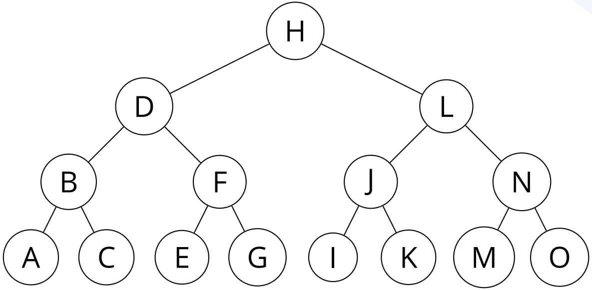
$n = 7$
$O() = 2.81 \Rightarrow 2$

$O(\log n)$ cont.

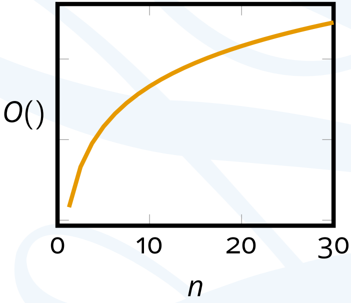
I

$O(\log n)$ complexity.

- Rate of increase gets lower and lower.
- $\log_2(100)$ is only 6.
- $\log_2(1000000000000)$ (trillion) is only 39.



$n = 15$
$O() = 3.91 \Rightarrow 3$

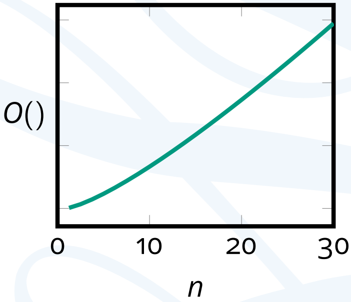


$O(n \log n)$

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 $O(n \log n)$.



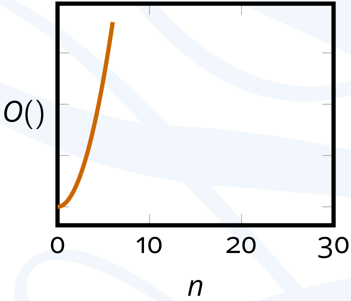
Introduction
Fibonacci example
Difficulty
Module content
Profiling
Efficiency
Optimization
$O()$ notation
Simple algorithms
Good algorithms
Bad algorithms
Recap

$O(2^n)$

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.



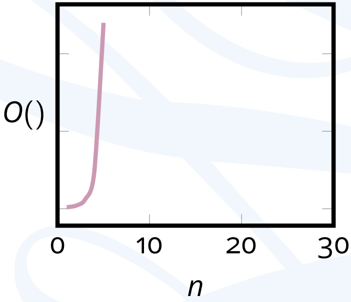
Introduction
Fibonacci example
Difficulty
Module content
Profiling
Efficiency
Optimization
$O()$ notation
Simple algorithms
Good algorithms
Bad algorithms
Recap

$O(n!)$

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.

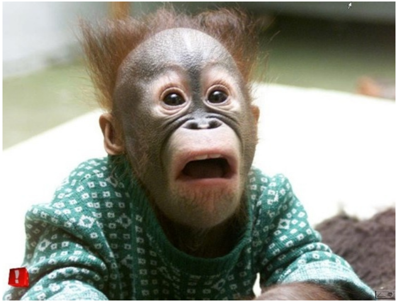


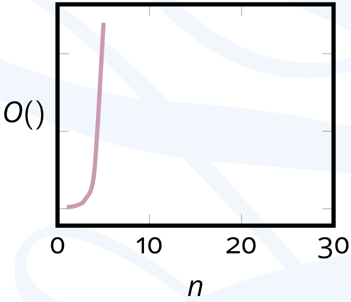
$O(n!)$

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.





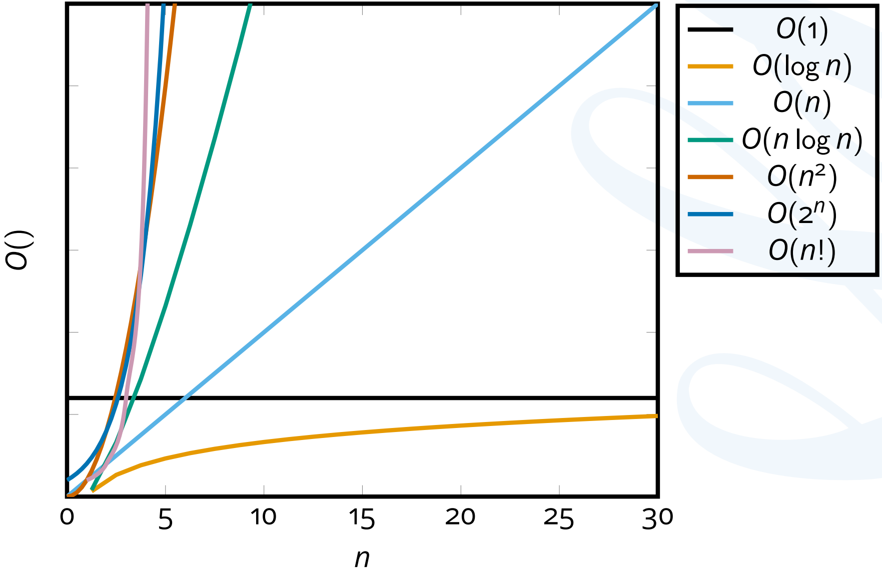
Scaling

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

		$n=2$	$n=10$	$n=100$
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

I

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.

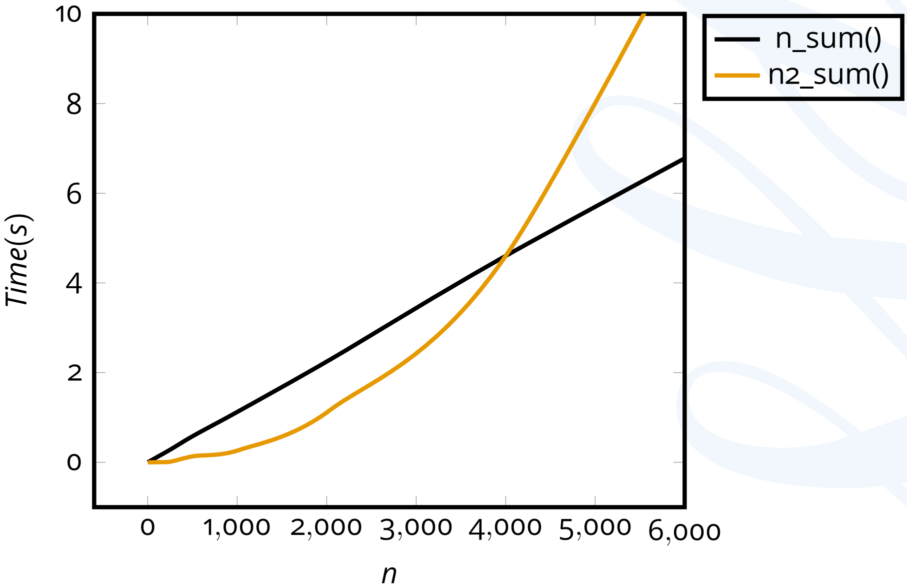
```python
def n_sum(sequence):
    total = 0
    for i in range(len(sequence)):
        total += sequence[i]
        time.sleep(0.001)
    return total
```

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



Why do I care?

C

- Everyone
  - Thinking algorithmically is critical programming skill.
  - Learning how to break down a problem into small steps.
    - Functional decomposition.
  - Evaluate algorithms.
    - Does this algorithm actually work?
  - Interview questions.
  - Without *O*() notation we can't discuss how algorithms compare.
  - Without *O*() can't discuss why some tasks are effectively impossible (travelling salesman).
- Ethical Hackers - *O*() important in discussing password security.
- Games Tech - *O*() explains the need for path finding and graphics work arounds.

Introduction

Fibonacci example

Difficulty

Module content

Profiling

Efficiency

Optimization

$O()$ notation

Simple algorithms

Good algorithms

Bad algorithms

Recap

Recap

- What is an algorithm.
- Code vs. algorithms.
- Heuristics = good enough solutions.
- $O()$ describes algorithm complexity in time and/or space.
- How your code *should* scale.
- $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!)$
- $< O(n^2)$ means polynomial.
- $\geq O(2^n)$ means exponential.
- Polynomial = easy problems.
- Non-polynomial = hard problems.

Introduction
Fibonacci example
Difficulty
Module content
Profiling
Efficiency
Optimization
$O()$ notation
Simple algorithms
Good algorithms
Bad algorithms
Recap

Expectations

C

- Complete the yellow Codio exercises for this week.
- Attempt the green Codio exercises for next week.
- If you have spare time attempt the red Codio exercises.
- If you are having issues come to the PSC.

  `https://gitlab.com/coventry-university/programming-support-lab/wikis/home`

The End