COMP2054-ACE: Introduction to big-Oh

ADE-Lec02a

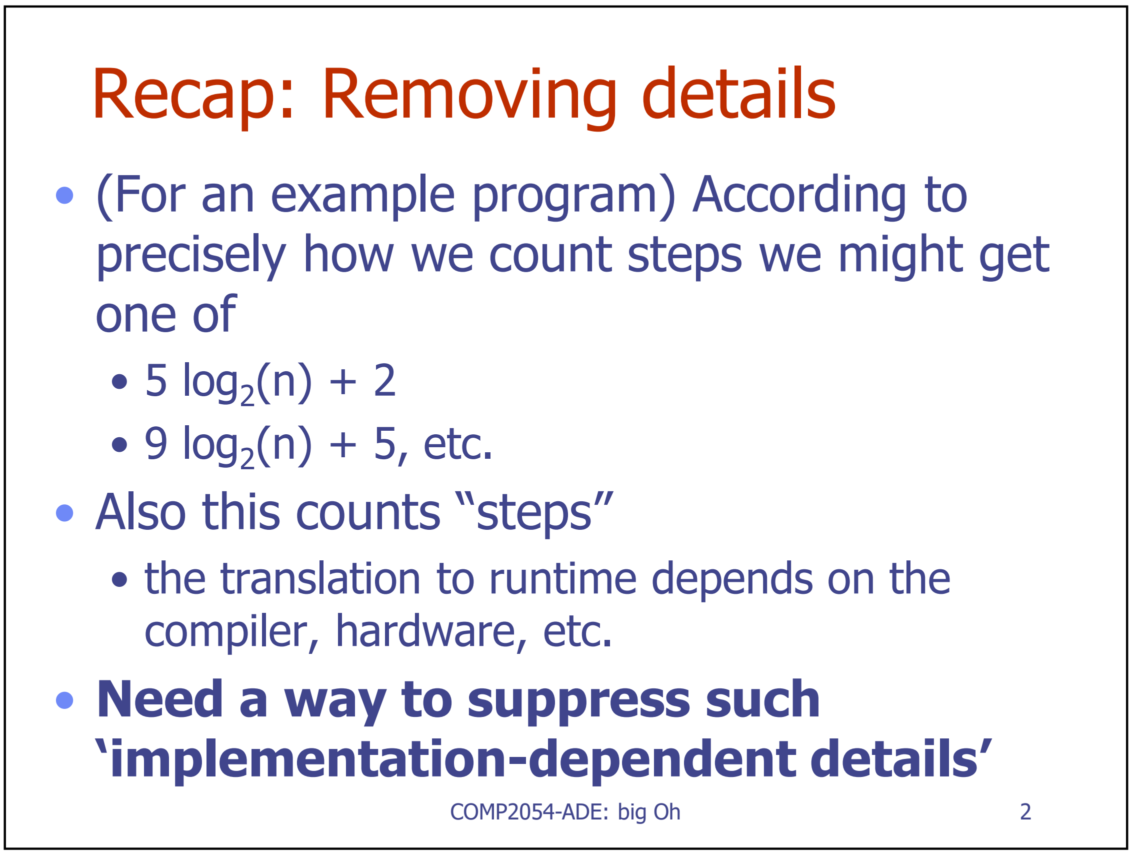
Lecturer: Andrew Parkes

Email: andrew.parkes@Nottingham.ac.uk

from http://www.cs.nott.ac.uk/~pszajp/

In this slides we give the motivations and context for the definition of big-Oh. The main point is that a proper understanding of a definition should include an understanding of the components of a definition.

Definitions are not things that exist of their own accord, but are designed by people to be useful to people.



Recap: Removing details

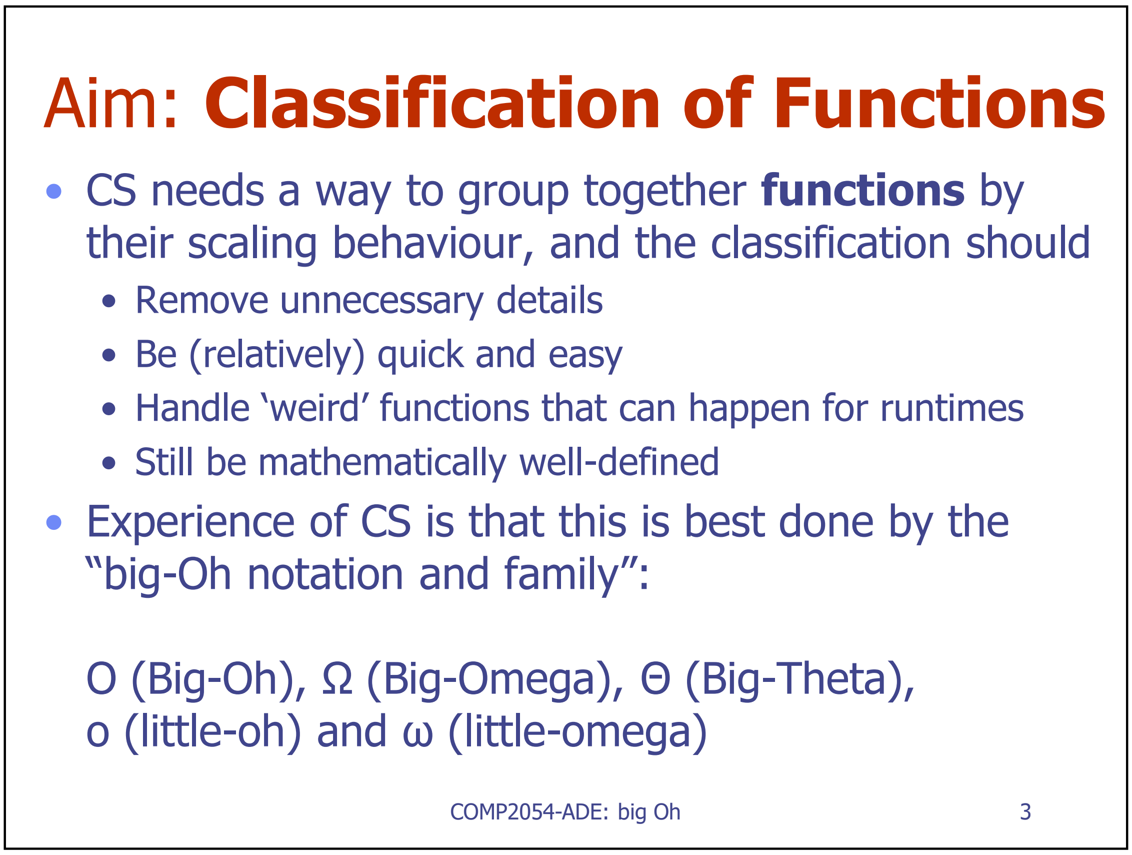
- (For an example program) According to precisely how we count steps we might get one of
  - $5 \log_2(n) + 2$
  - $9 \log_2(n) + 5$, etc.
- Also this counts “steps”
  - the translation to runtime depends on the compiler, hardware, etc.
- **Need a way to suppress such ‘implementation-dependent details’**

As we saw in the last lecture, we can do some counting of primitive operations.

But there is no unique way to count and so the constants that occur might well vary between methods.

Differences can arise because of different compilers, hardware etc.

So we need to formal way suppress the details and pick on the log of n.



Aim: **Classification of Functions**

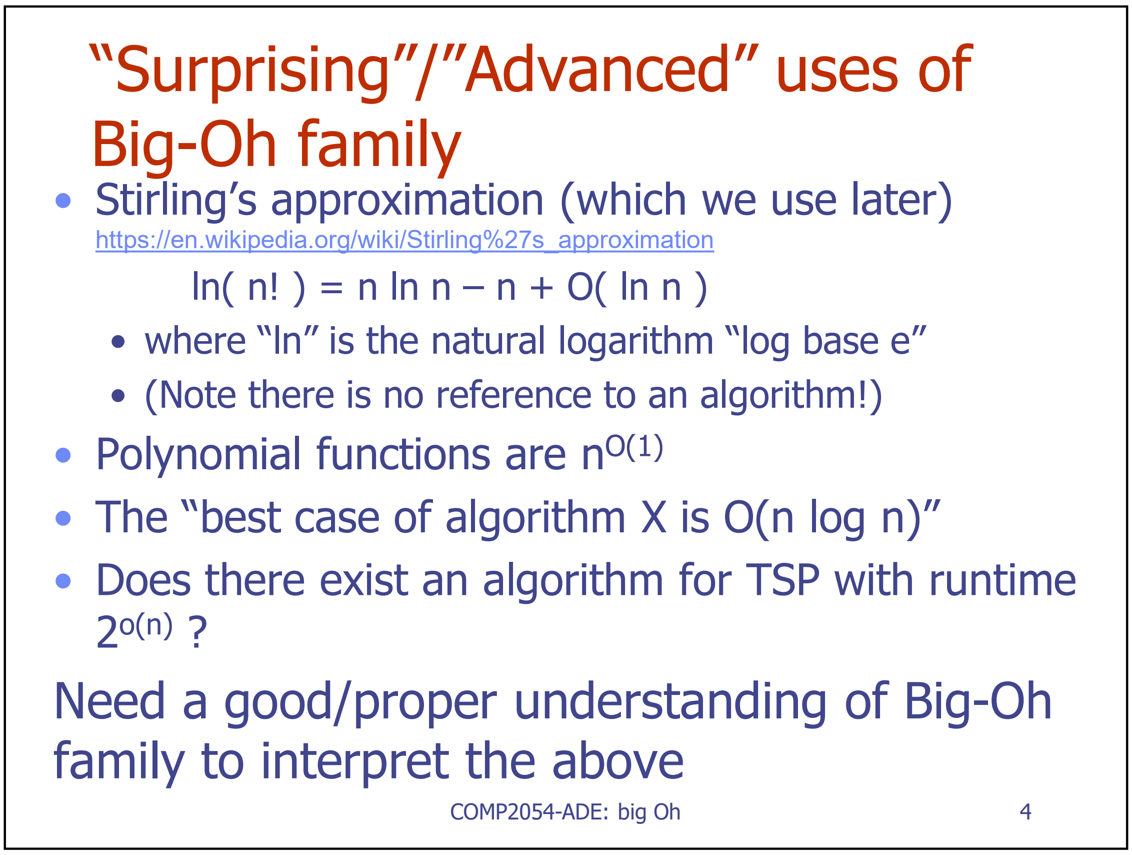
- CS needs a way to group together **functions** by their scaling behaviour, and the classification should
  - Remove unnecessary details
  - Be (relatively) quick and easy
  - Handle 'weird' functions that can happen for runtimes
  - Still be mathematically well-defined
- Experience of CS is that this is best done by the "big-Oh notation and family":

  O (Big-Oh), Ω (Big-Omega), Θ (Big-Theta),
  o (little-oh) and ω (little-omega)

Hence, in CS we need a way to classify functions in ways that are most useful for CS.

Firstly, notice here that I deliberately write "**FUNCTIONS**" and not algorithms. The approach is to consider the functions that arise from analysis of runtimes of algorithms,

but then forget about the algorithms and instead focus on grouping together functions in ways that put functions into sensible classes.



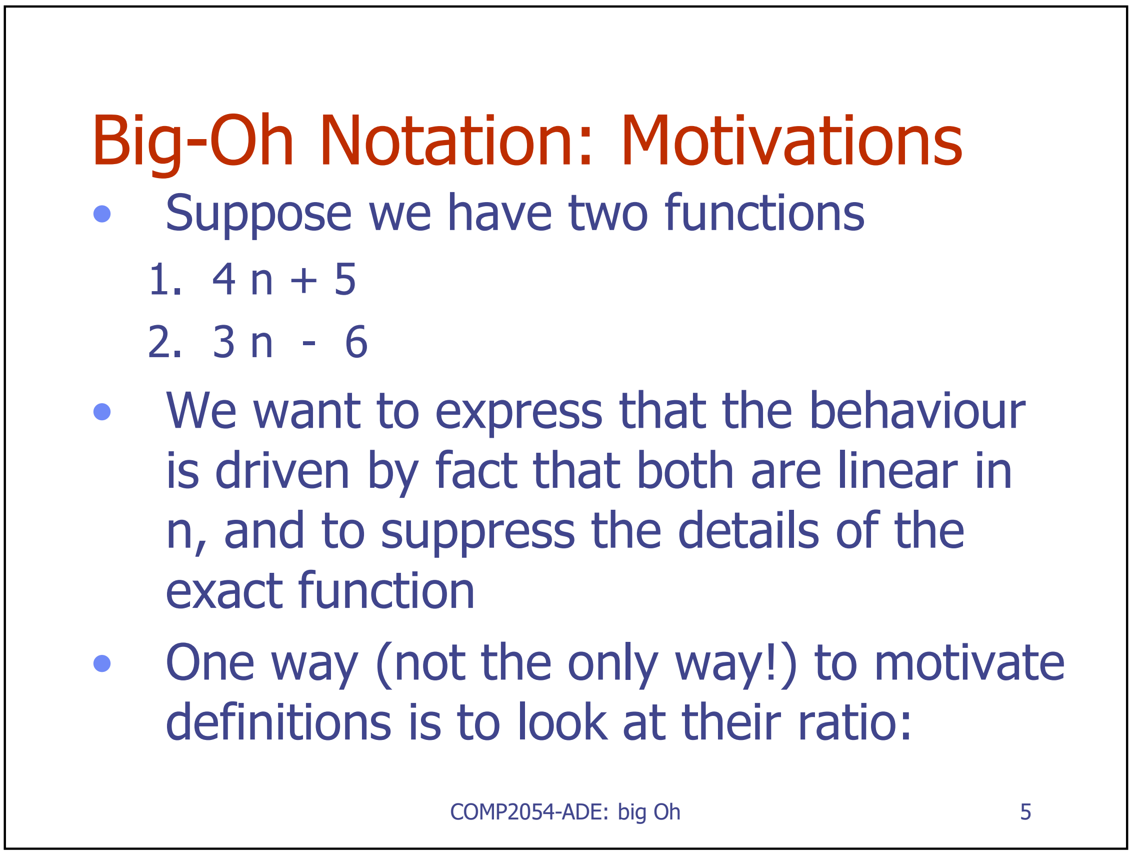
"Surprising"/"Advanced" uses of Big-Oh family

- Stirling's approximation (which we use later)
  https://en.wikipedia.org/wiki/Stirling%27s_approximation
  $$\ln(n!) = n \ln n - n + O(\ln n)$$
  - where "ln" is the natural logarithm "log base e"
  - (Note there is no reference to an algorithm!)
- Polynomial functions are $n^{O(1)}$
- The "best case of algorithm X is O(n log n)"
- Does there exist an algorithm for TSP with runtime $2^{o(n)}$?

Need a good/proper understanding of Big-Oh family to interpret the above

This is just more motivation, especially if you have seen the standard usage of big-Oh as a "worst case of the runtime of an algorithm".

It shows that it has usages even when we are only talking about functions. This is because the big-Oh family are about functions, and their usage on runtimes of algorithms is just one application.

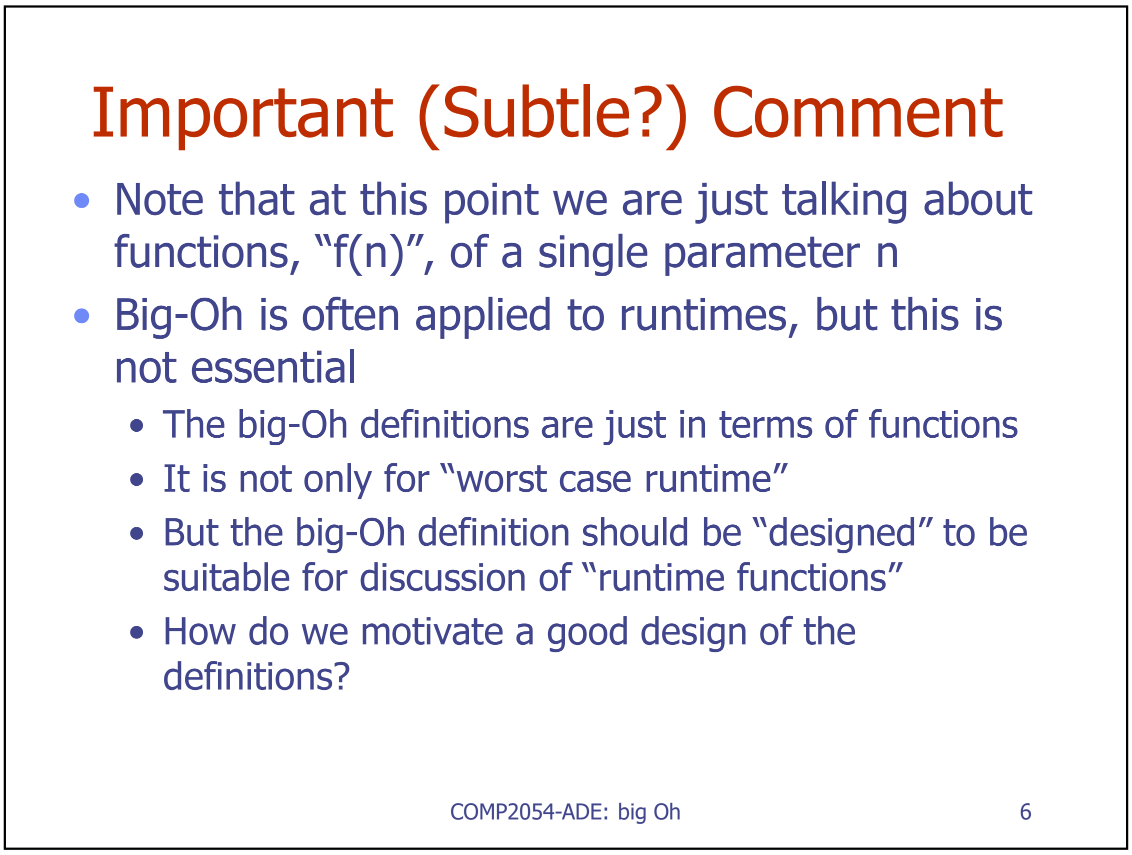


Big-Oh Notation: Motivations

- Suppose we have two functions
  1. 4 n + 5
  2. 3 n - 6
- We want to express that the behaviour is driven by fact that both are linear in n, and to suppress the details of the exact function
- One way (not the only way!) to motivate definitions is to look at their ratio:

Let's consider a simpler example, and take two linear functions.
Obviously they both are linear in n, but we need a formal way that allows to capture this, and also many other cases.

A natural way is to start by looking at their ratio.



Important (Subtle?) Comment

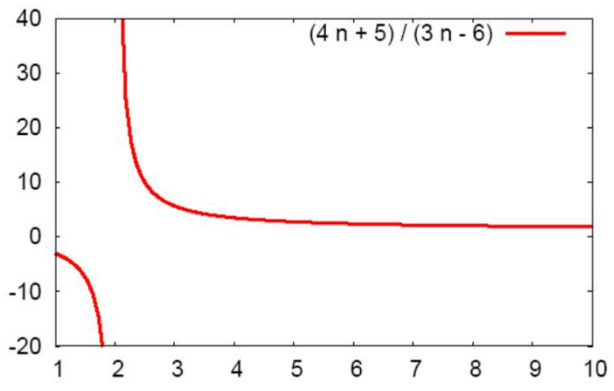
- Note that at this point we are just talking about functions, “f(n)”, of a single parameter n
- Big-Oh is often applied to runtimes, but this is not essential
  - The big-Oh definitions are just in terms of functions
  - It is not only for “worst case runtime”
  - But the big-Oh definition should be “designed” to be suitable for discussion of “runtime functions”
  - How do we motivate a good design of the definitions?

Before doing so, I just emphasis again that we are now just talking about a function of a single parameter.

We will soon arrive at Big-Oh, and it is usually applied to worst run-times, but it is independent of that is it is really just capturing properties of functions.
It can have other uses.

However, we do need a system of describing functions that is tailored to be useful in CS and efficiency.

Ratio of two linear functions

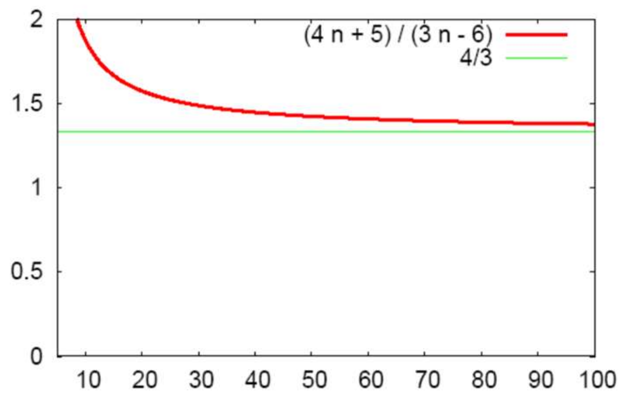


- The behaviour at small ***n*** is messy

If we consider the ratio as a function of n, then the first observation is that the behaviour at small n is a bit of a mess.

We can get rid of this by just looking past the mess, and considering just values of the ratio past some value of n.

Ratio of two linear functions



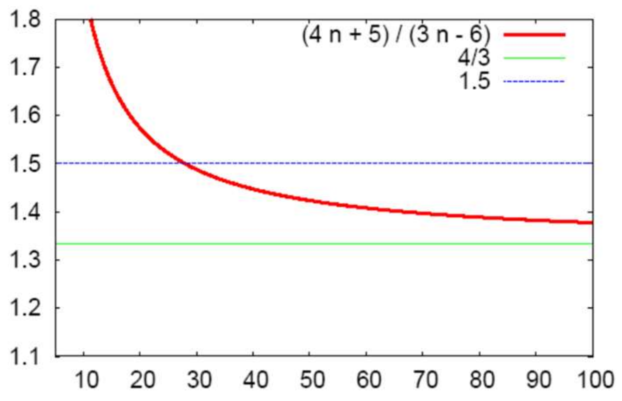
- At larger values of ***n*** the ratio starts to behave more predictably
    - Suggests definitions should use “forall n >= …”

If we focus on n at least 8, then it becomes all much cleaner.
Looking at the graph, then it is clear that it gets ever closer to 4/3.
When n is very large, then we can ignore the “+5” and the “-6” and then the ratio is precisely 4/3.

However, note that at any finite value of n, the ratio is still above n, even though it gets close.

How can we capture this?

Ratio of two linear functions



- It looks like:
  $(4n+5) \leq 1.5 * (3n-6)$ for all $n \geq 30$?

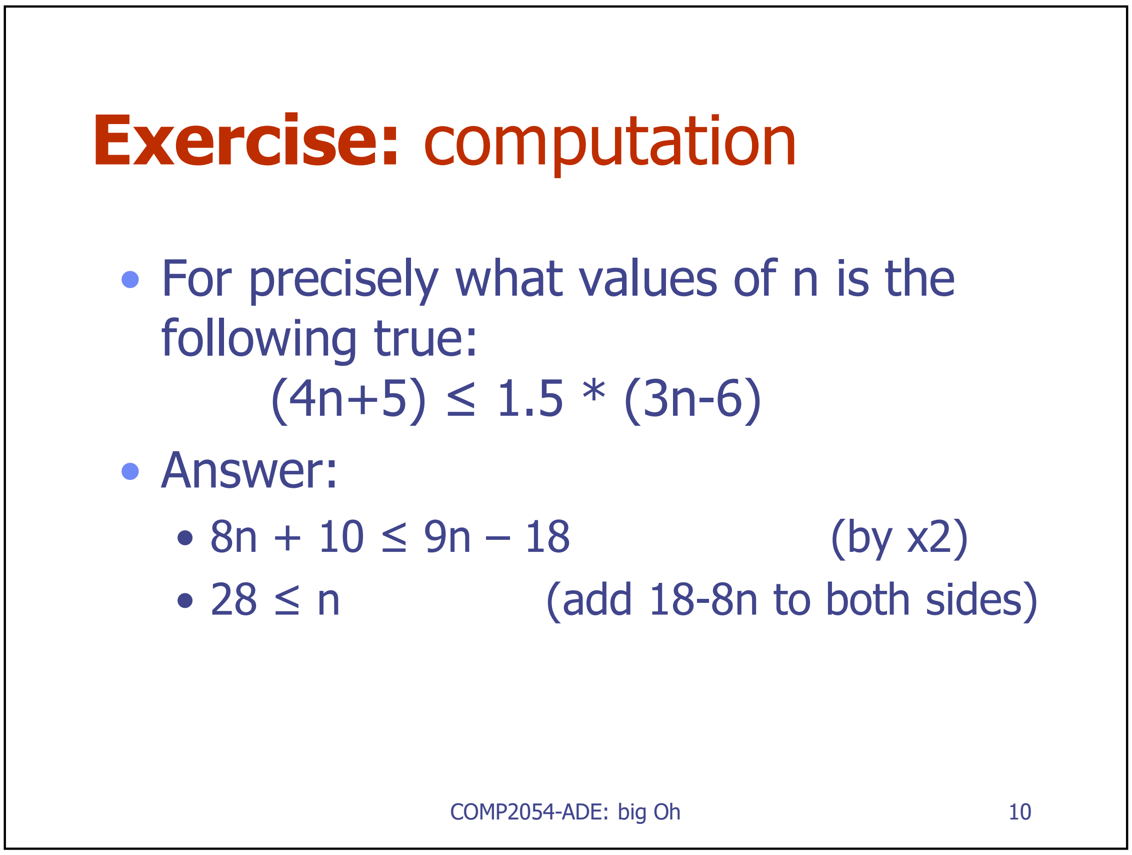
It is natural to then just say that it gets within some fixed factor.
Suppose that we pick 1.5

This choice is arbitrary – any number strictly bigger than 1 would do.

Then just reading from the graph we can make the statement that the one function is less than 1.5 times the other,

At least once n is above some somewhere around n=30.

Of course, we can tighten this up with some algebra.



Exercise: computation

- For precisely what values of n is the following true:

  $(4n+5) \leq 1.5 * (3n-6)$
- Answer:
  - $8n + 10 \leq 9n - 18$ (by x2)
  - $28 \leq n$ (add 18-8n to both sides)

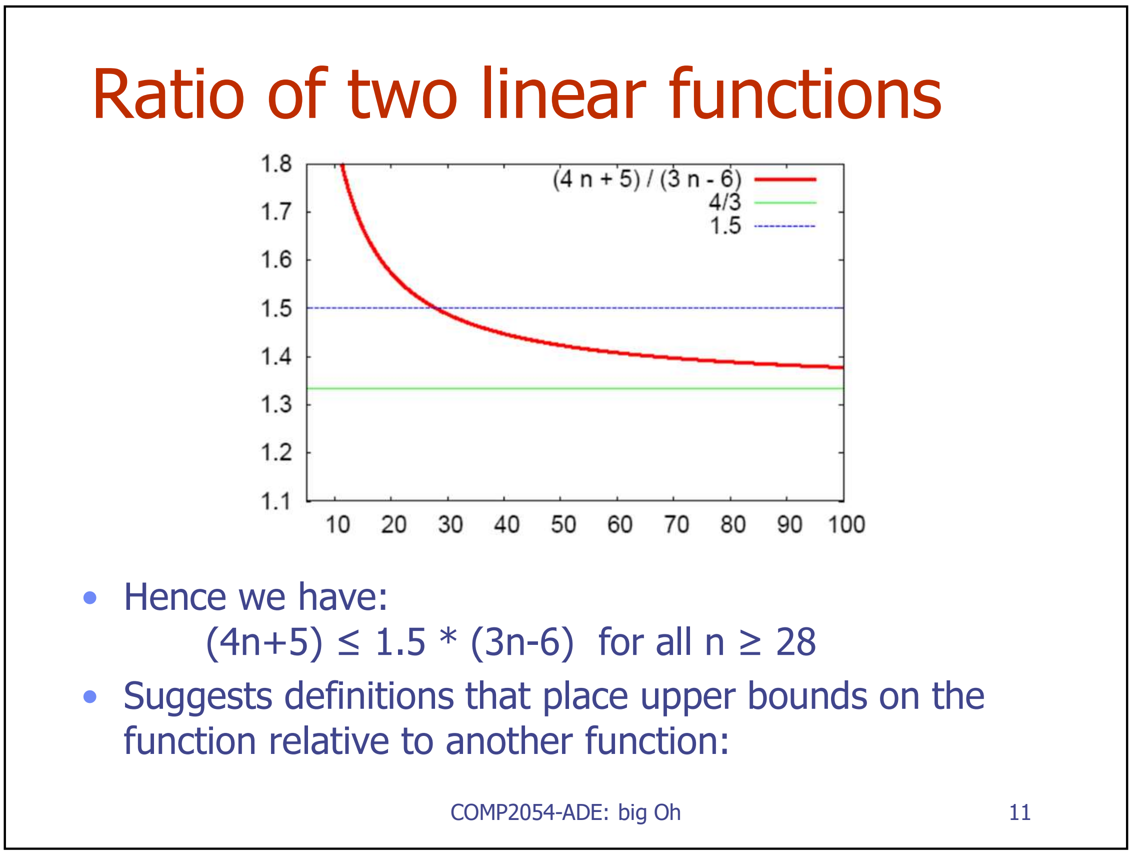
This is just some straightforward algebra – but is of the kind that will be needed. Suppose we just to know the value of n at which the ratio becomes less than or equal to 1.5.

Just need to solve the equation.
This is standard maths.

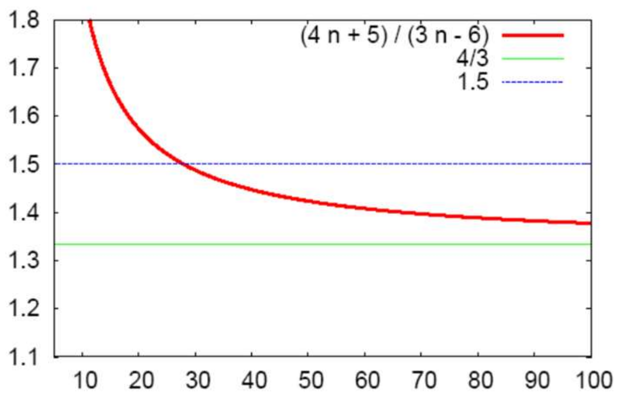
Multiply up by two and expand the brackets.
Then move entries.
A straightforward piece of maths.

ASIDE:

But include it as you might find it useful exercise to think whether or not you would be able to this computation, or something similar in Lean.
Probably want to change the "1.5" to "2" to avoid going outside of nat. Might want to watch out for the "minus".



Ratio of two linear functions



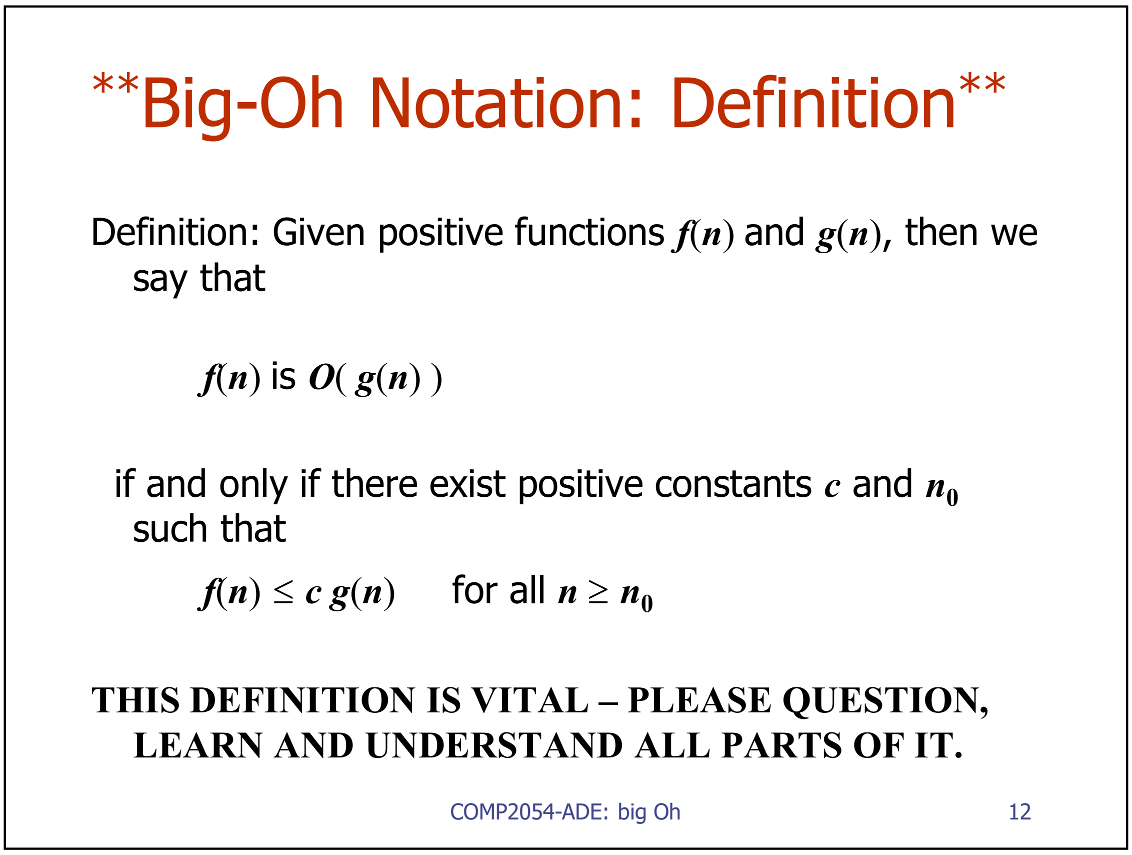
- Hence we have:
  $(4n+5) \leq 1.5 * (3n-6)$ for all $n \geq 28$
- Suggests definitions that place upper bounds on the function relative to another function:

COMP2054-ADE: big Oh 11

So then we have that we relate the two functions in terms of their being a relation in that one is no more than a constant times the other, at least once we are above a certain value.

Standard style in maths is then to take the general structure and capture the idea in a general definition.

Which takes us fairly directly to the definition of big-Oh



Big-Oh Notation: Definition

Definition: Given positive functions $f(n)$ and $g(n)$, then we say that

$f(n)$ is $O(\, g(n) \,)$

if and only if there exist positive constants c and n_0 such that

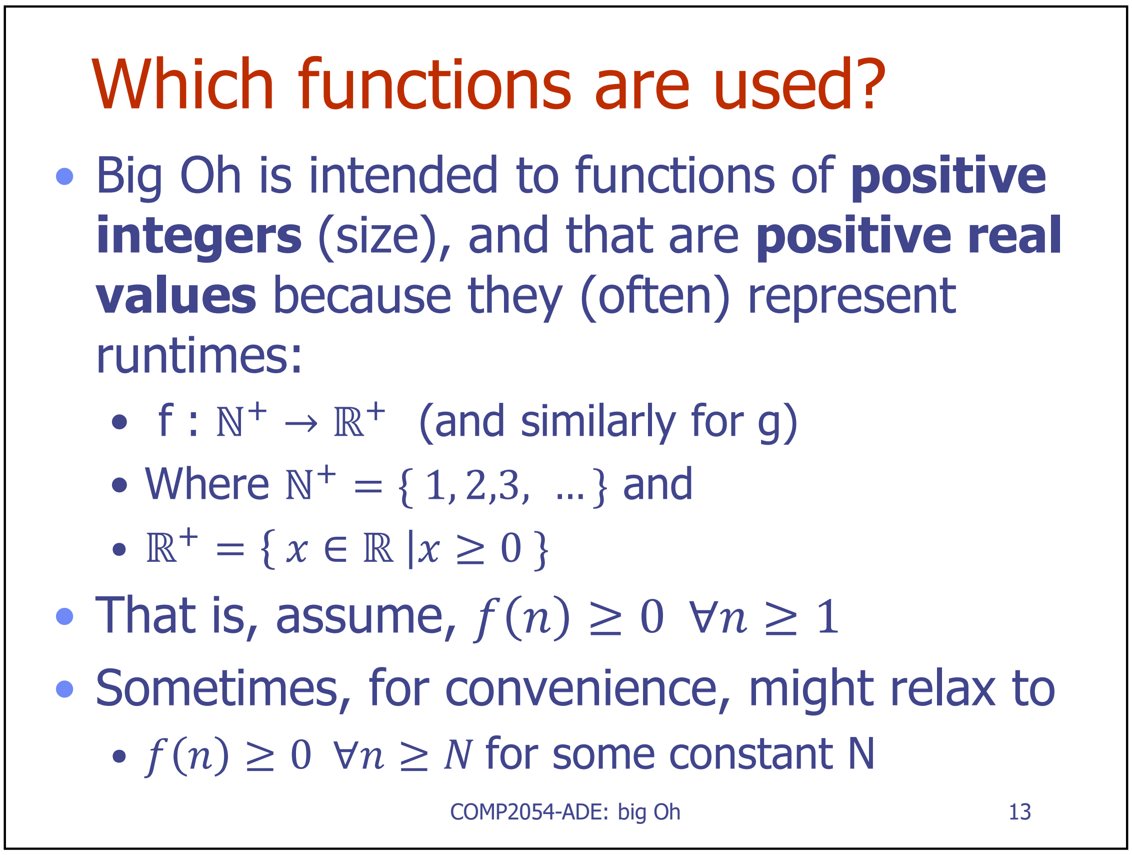
$f(n) \leq c\, g(n)$ for all $n \geq n_0$

THIS DEFINITION IS VITAL – PLEASE QUESTION, LEARN AND UNDERSTAND ALL PARTS OF IT.

COMP2054-ADE: big Oh 12

Hence, the natural definition that captures the example is to consider two functions, and say that one is Big-Oh of the other given the conditions.

The condition is that there are numbers c and n0 that mean that f is no more than c times g, as soon as n is at least n0.

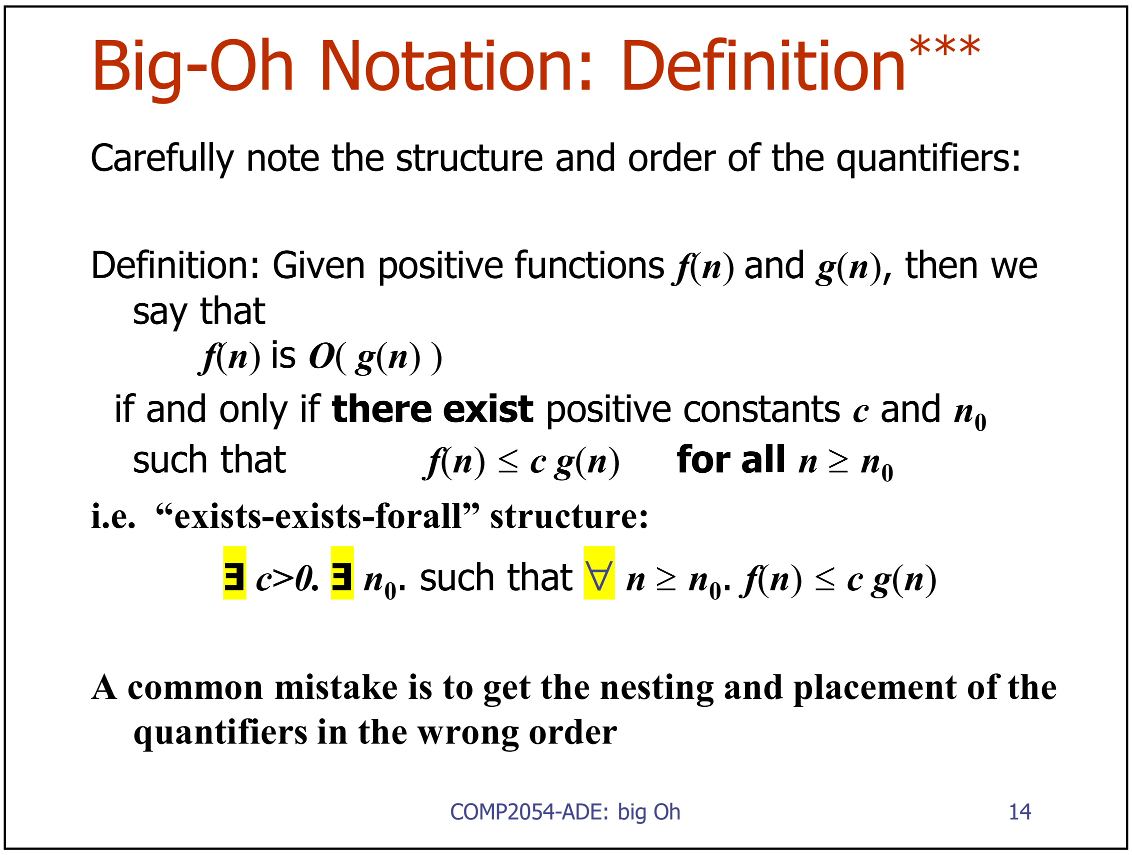


Which functions are used?

- Big Oh is intended to functions of **positive integers** (size), and that are **positive real values** because they (often) represent runtimes:
  - f : $\mathbb{N}^+ \rightarrow \mathbb{R}^+$ (and similarly for g)
  - Where $\mathbb{N}^+ = \{ 1, 2,3, \ \dots \}$ and
  - $\mathbb{R}^+ = \{ x \in \mathbb{R} \,| x \geq 0 \}$
- That is, assume, $f(n) \geq 0 \ \ \forall n \geq 1$
- Sometimes, for convenience, might relax to
  - $f(n) \geq 0 \ \ \forall n \geq N$ for some constant N

Since we are thinking of algorithms then the argument is going to be sizes of problems, and may want to restrict to positive integers.
We want the value of the function to be usable for runtimes, and so we probably want to use it as a real number (or at least floating point).

Might want to convert functions to be non-negative.
However, such issues usually just appear at “small n” and can usually ignore.



Big-Oh Notation: Definition***

Carefully note the structure and order of the quantifiers:

Definition: Given positive functions $f(n)$ and $g(n)$, then we say that

$f(n)$ is $O(g(n))$

if and only if **there exist** positive constants c and n_0 such that $f(n) \leq c\, g(n)$ **for all** $n \geq n_0$

i.e. "exists-exists-forall" structure:

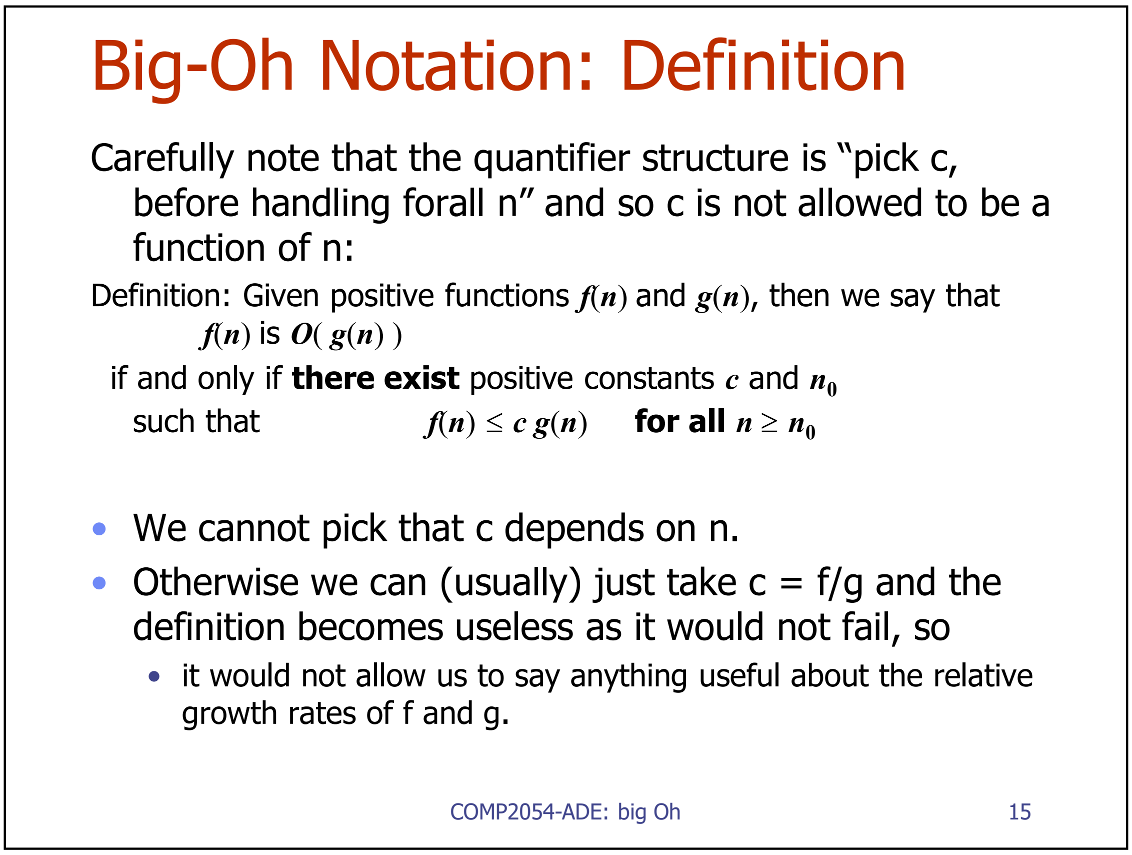
$\exists\ c>0.\ \exists\ n_0.$ such that $\forall\ n \geq n_0.\ f(n) \leq c\, g(n)$

A common mistake is to get the nesting and placement of the quantifiers in the wrong order

One of the most common mistakes is to accidentally do something that corresponds to swapping the order of the quantifiers.

As you will have found in doing proofs in lean, it is not at all allowed to just swap quantifiers.

"forall exists" is quite different from "exists forall" and totally changes the meaning.
Be careful not to do this. It usually happens implicitly rather than explicitly.
I.e. some argument would only work if the order is swapped.
This is a common problem if using natural language – and an everyday way of speaking/writing.



Big-Oh Notation: Definition

Carefully note that the quantifier structure is "pick c, before handling forall n" and so c is not allowed to be a function of n:

Definition: Given positive functions $f(n)$ and $g(n)$, then we say that

$f(n)$ is $O(g(n))$

if and only if **there exist** positive constants c and n_0

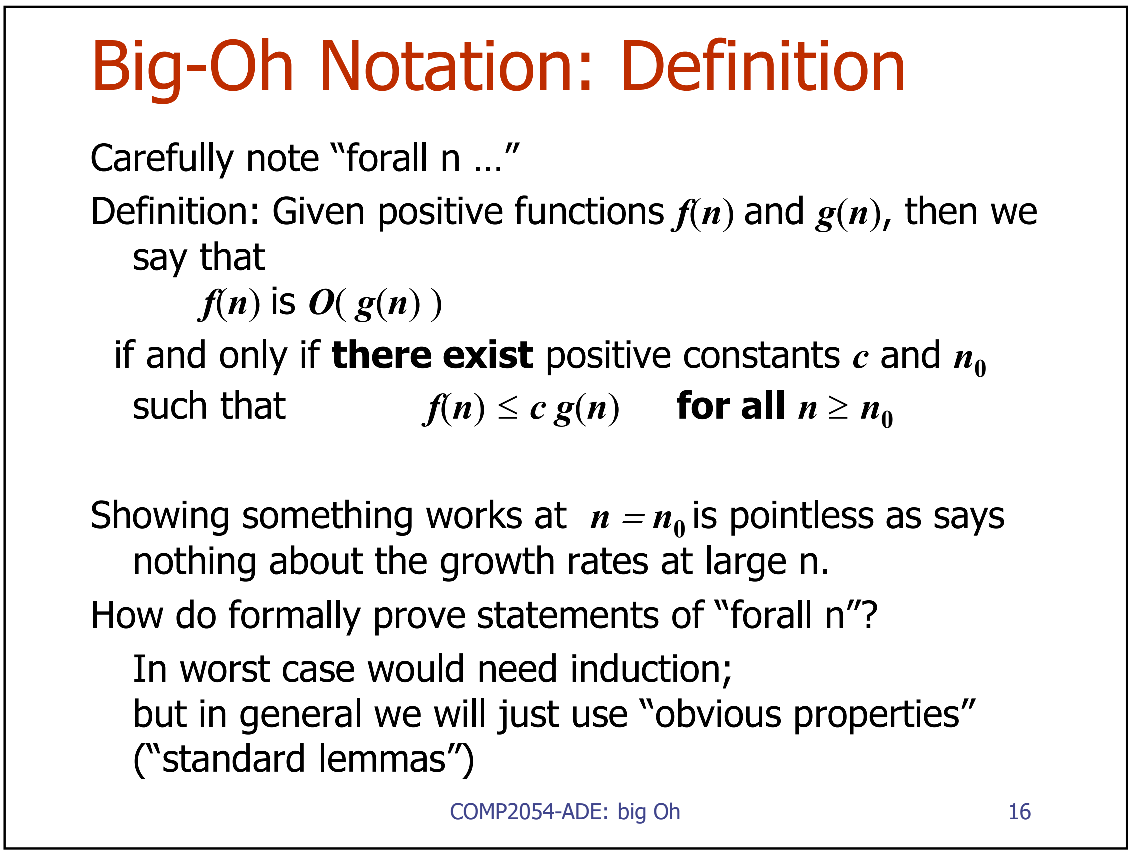
such that $f(n) \leq c\, g(n)$ **for all** $n \geq n_0$

- We cannot pick that c depends on n.
- Otherwise we can (usually) just take c = f/g and the definition becomes useless as it would not fail, so
  - it would not allow us to say anything useful about the relative growth rates of f and g.

One of the most common mistakes is also to do something that corresponds to picking a function for c, and allowing it do depend on n.

We could write a definition that allows this; but it would be essentially useless.

It is vital that c does not depend on n, as we want to use the big-Oh language to compare growth rates of functions.



Big-Oh Notation: Definition

Carefully note "forall n …"

Definition: Given positive functions $f(n)$ and $g(n)$, then we say that

$f(n)$ is $O(g(n))$

if and only if **there exist** positive constants c and n_0 such that $f(n) \le c\, g(n)$ **for all** $n \ge n_0$

Showing something works at $n = n_0$ is pointless as says nothing about the growth rates at large n.

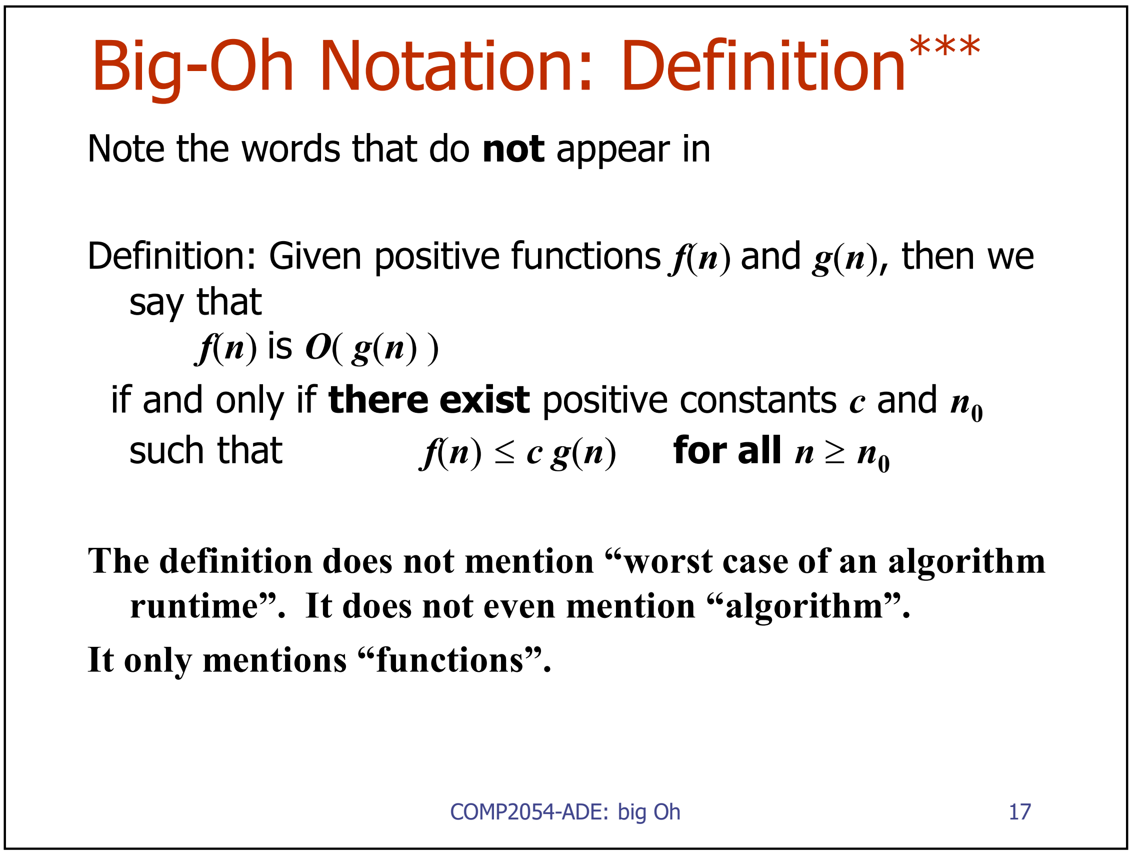
How do formally prove statements of "forall n"?

In worst case would need induction;
but in general we will just use "obvious properties" ("standard lemmas")

One of the most common mistakes is to accidentally do something that corresponds to swapping the order of the quantifiers.

As you will have found in doing proofs in lean, it is not at all allowed to just swap quantifiers.

"forall exists" is quite different from "exists forall" and totally changes the meaning.
Be careful not to do this. It usually happens implicitly rather than explicitly.
I.e. some argument would only work if the order is swapped.
This is a common problem if using natural language – and an everyday way of speaking/writing.



Big-Oh Notation: Definition***

Note the words that do **not** appear in

Definition: Given positive functions $f(n)$ and $g(n)$, then we say that

$f(n)$ is $O(\, g(n)\,)$

if and only if **there exist** positive constants c and n_0 such that $f(n) \leq c\, g(n)$ **for all** $n \geq n_0$

The definition does not mention "worst case of an algorithm runtime". It does not even mention "algorithm".

It only mentions "functions".

Personally, one of the hardest things in the IFR part was to undo the ingrained and "automatic" usage of the "excluded middle".

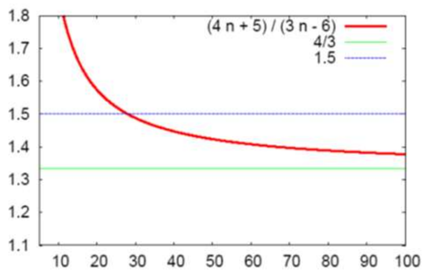
E.g. it can require a

Part of the learning was to undo a way of thinking about logic, and so build a new (better?) way of thinking.

This may seems like "going backwards" and so "bad" but it is actually good as ultimately it leads to much more powerful ways of thinking.

It is similar with Big-Oh, that once have a split between "functions" and "algorithms" then a lot more can be done.

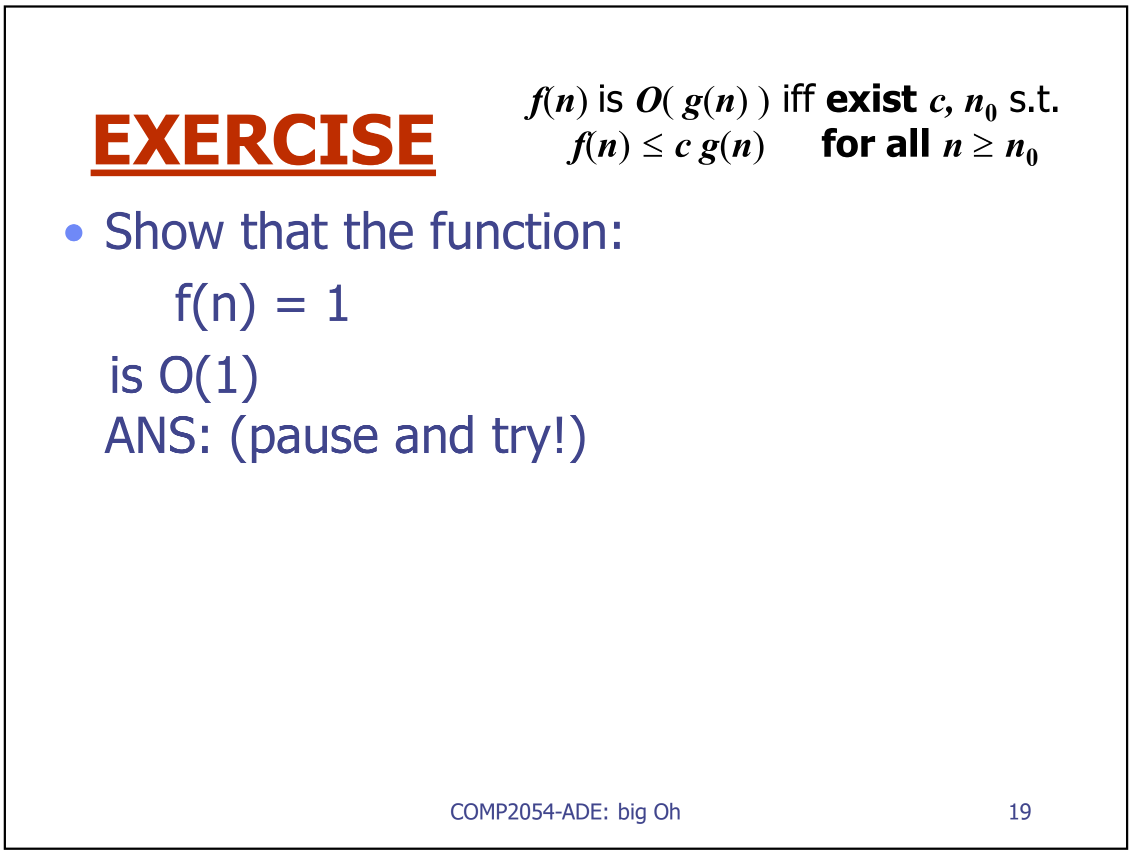
Ratio of two linear functions



- Since: $(4n+5) \leq 1.5 * (3n-6)$ for all $n \geq 28$
- We now have: $(4n+5)$ is $O((3n-6))$
  - Using $c=1.5$ and $n_0=28$

Going back to the example, we can now rephrase the way of writing this.
We show that the one function is big-Oh of the other. The proof we gave used c=1.5 and n0=28.

So this is our first big-Oh question.



EXERCISE

$f(n)$ is $O(\ g(n)\ )$ iff **exist** $c,\ n_0$ s.t.
$f(n) \leq c\ g(n)$ **for all** $n \geq n_0$

- Show that the function:

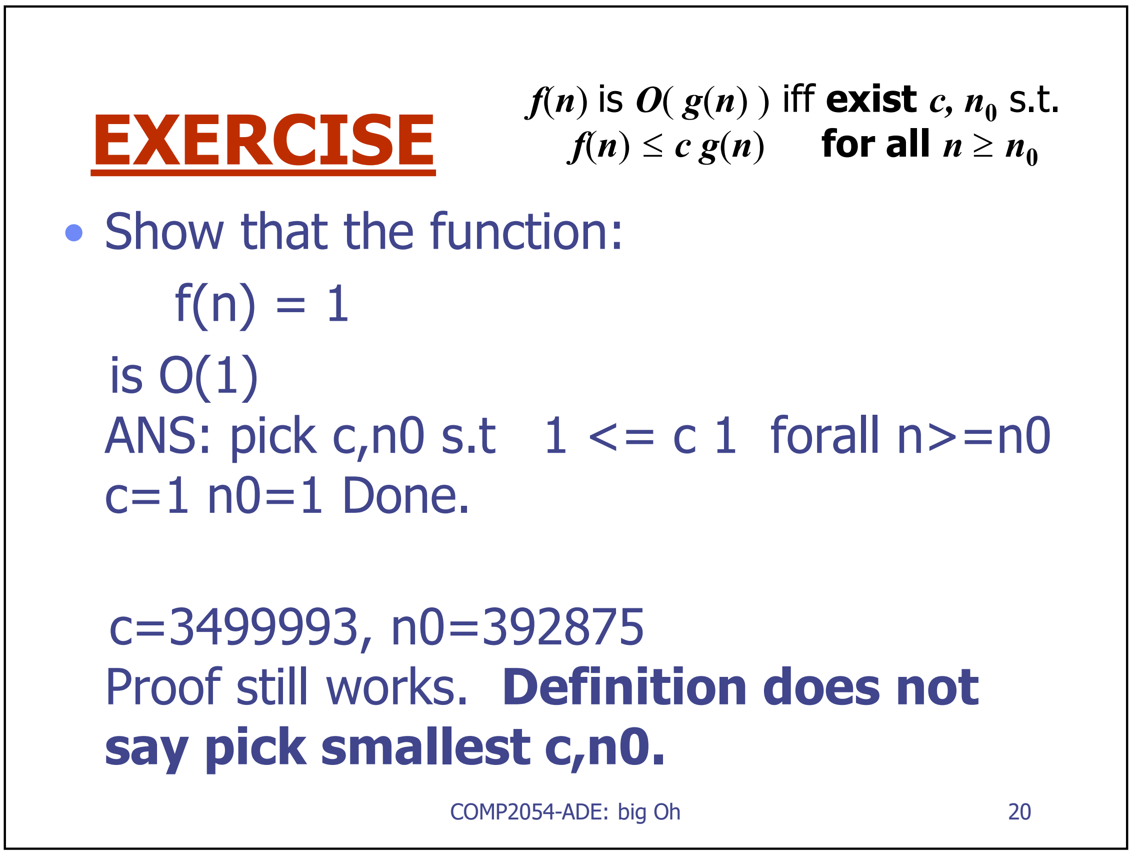
  $f(n) = 1$

  is $O(1)$

  ANS: (pause and try!)

We shall follow good practice and start with the most trivial case.

If working offline – then try this yourself first!!



EXERCISE

$f(n)$ is $O(g(n))$ iff **exist** c, n_0 s.t.
$f(n) \le c\,g(n)$ **for all** $n \ge n_0$

- Show that the function:

  f(n) = 1

  is O(1)

  ANS: pick c,n0 s.t 1 <= c 1 forall n>=n0
  c=1 n0=1 Done.

  c=3499993, n0=392875
  Proof still works. **Definition does not say pick smallest c,n0.**

We shall follow good practice and start with the most trivial case.
Not that a constant is just a function as well, so taking f and g to be '1' is perfectly allowed.
Can we prove that this is true?

FIRST STEP: identify f and g and directly substitute them into the definition.
THAT IS: write out the definition in full, after the substitution

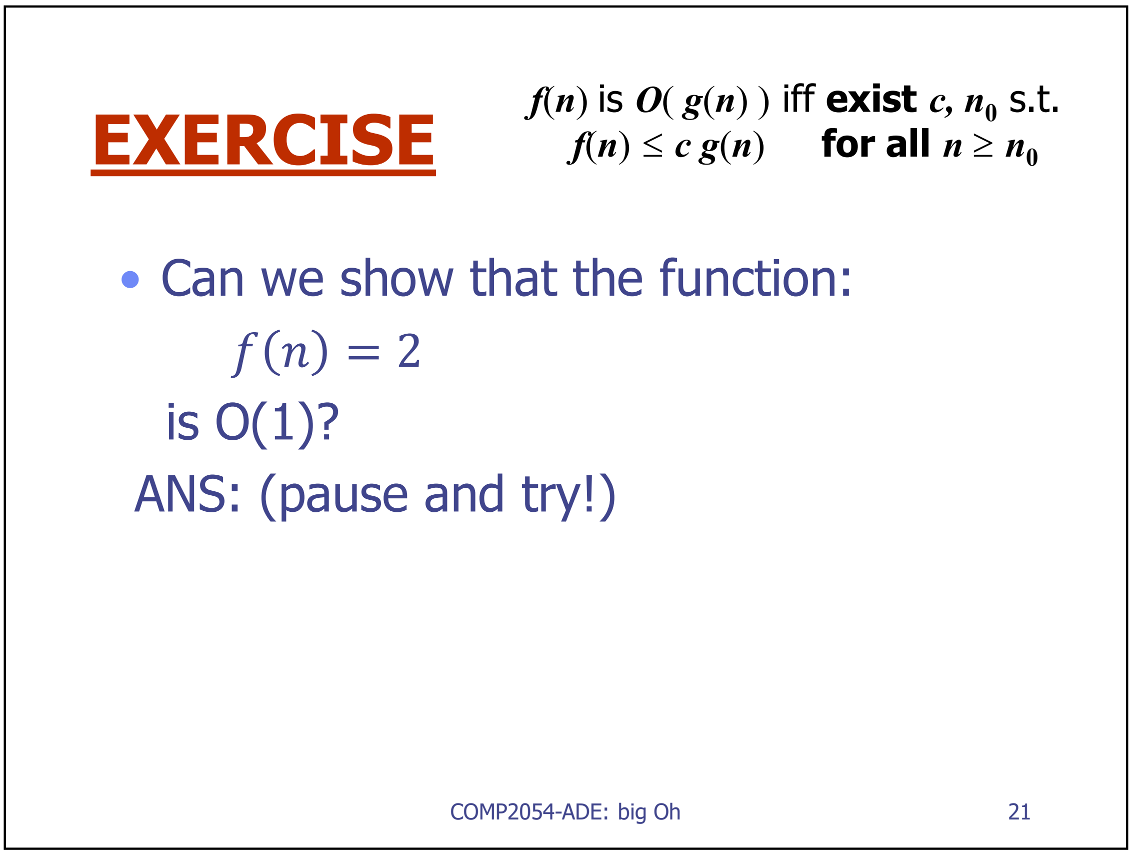
Do there exist c and n0 s.t. 1 <= c . 1 for all n >= n0.

Hence, we can pick any value of c that is bigger than 1.

Picking c=1 is natural.
Picking c = 2^100 is formally allowed, but would be considered as bad style – it would just raise the question of why such an unreasonable number was selected.

Also, we ignored n0, but formally we should give a value, e.g. n0=1.

Again other higher values of n0 would also work.
Note that if some value of n0 works, then trivially any higher value of n0 also works.



EXERCISE

$f(n)$ is $O(\ g(n)\ )$ iff **exist** c, n_0 s.t.
$f(n) \leq c\ g(n)$ **for all** $n \geq n_0$

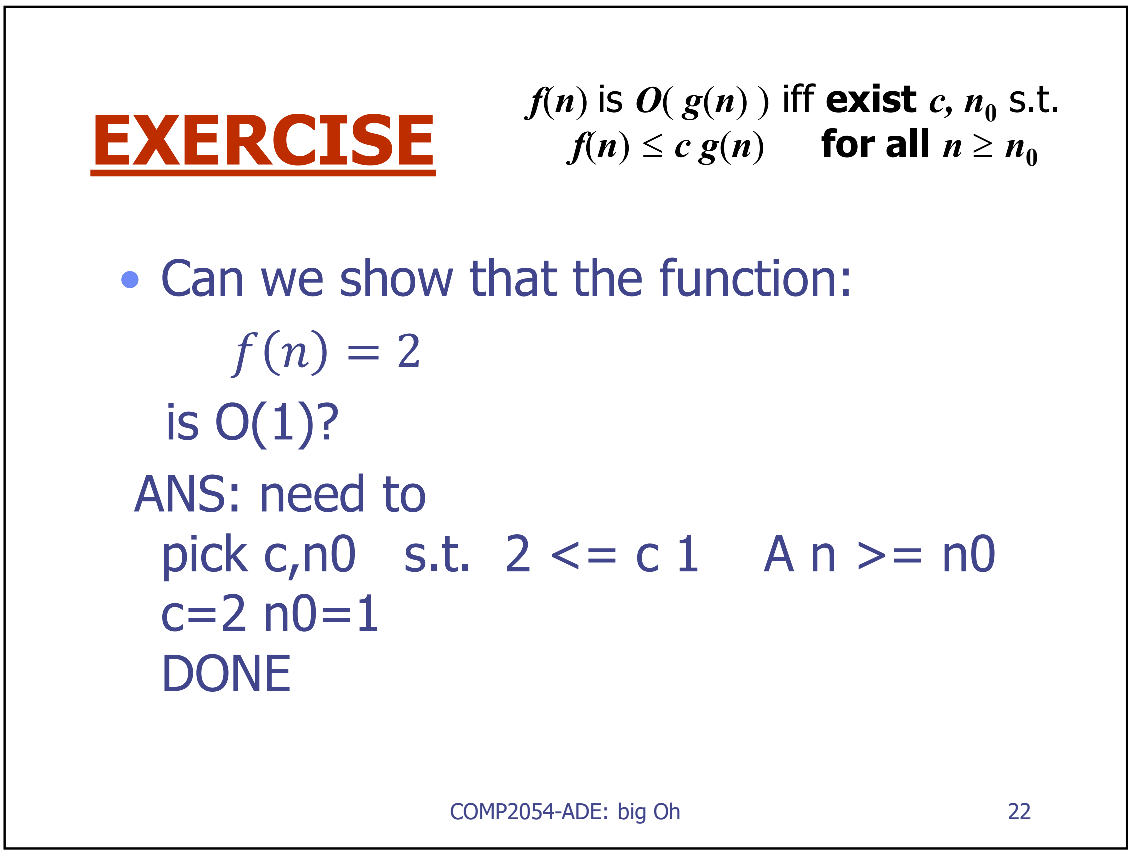
- Can we show that the function:

  $f(n) = 2$

  is O(1)?

ANS: (pause and try!)

Basically the same as before.



EXERCISE

$f(n)$ is $O(g(n))$ iff **exist** c, n_0 s.t. $f(n) \leq c\, g(n)$ **for all** $n \geq n_0$

- Can we show that the function:

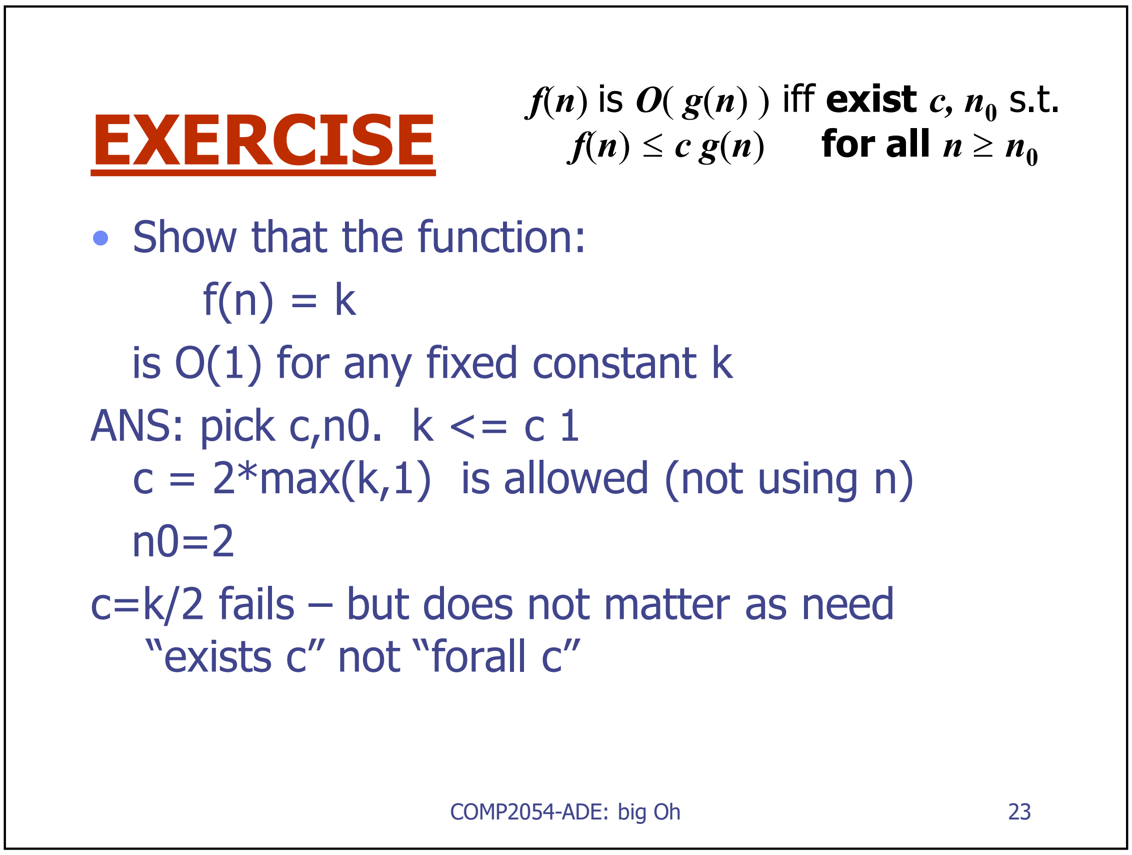
  $f(n) = 2$

  is O(1)?

ANS: need to
  pick c,n0 s.t. 2 <= c 1 A n >= n0
  c=2 n0=1
  DONE

COMP2054-ADE: big Oh 22

Basically the same as before.



EXERCISE

$f(n)$ is $O(g(n))$ iff **exist** c, n_0 s.t. $f(n) \le c\, g(n)$ **for all** $n \ge n_0$

- Show that the function:

  f(n) = k

  is O(1) for any fixed constant k

ANS: pick c,n0. k <= c 1

c = 2*max(k,1) is allowed (not using n)

n0=2

c=k/2 fails – but does not matter as need "exists c" not "forall c"

Slightly different version. What about something like f(n)=3?

We can pick any number k, as long as it is fixed.

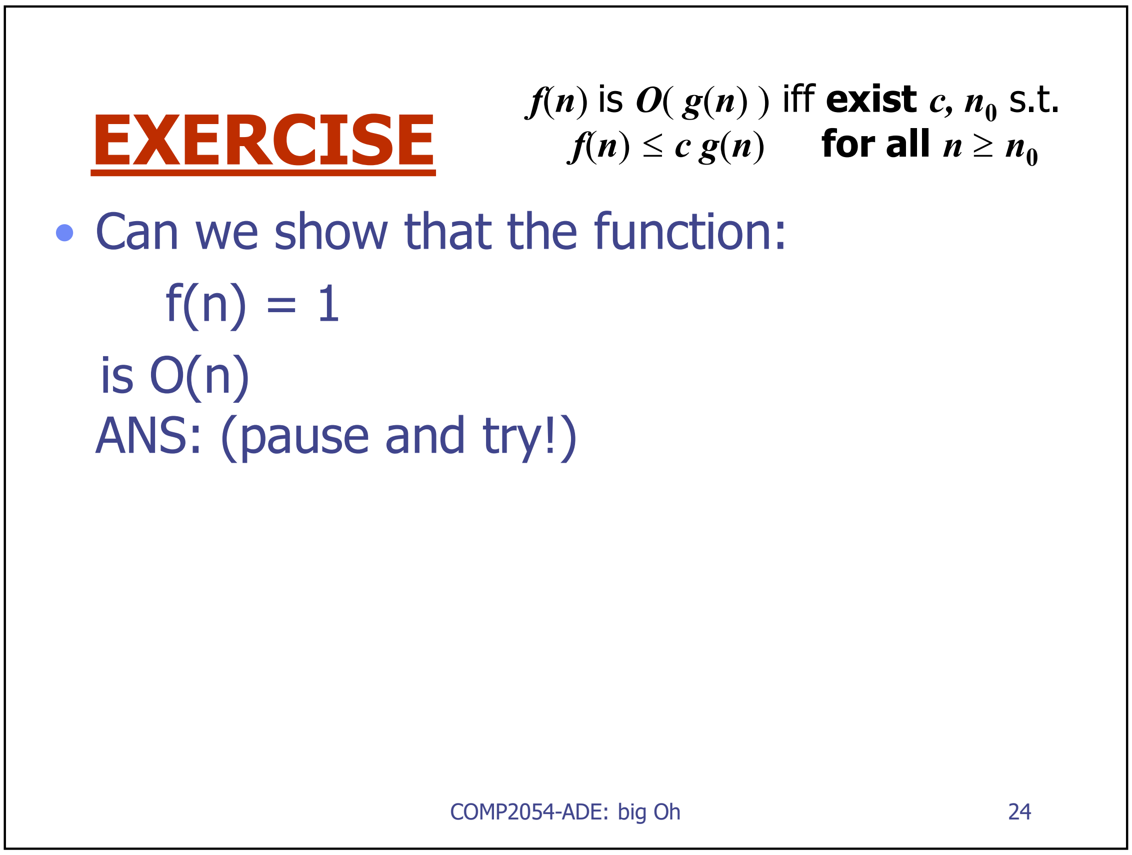
(In such cases, if this is confusing, then can just start by picking some number, and doing the proof, and then later try to make it more general).

We then just need k <= c . 1 forall n >= n0
Hence we can pick c=k when k is positive.

If k were negative then it is not really the kind of function we want to consider,

But could formally take some positive value for c instead.

Again, formally, we also need to make some choice for n0 even though nothing depends on n.



EXERCISE

$f(n)$ is $O(g(n))$ iff **exist** c, n_0 s.t. $f(n) \leq c\, g(n)$ **for all** $n \geq n_0$

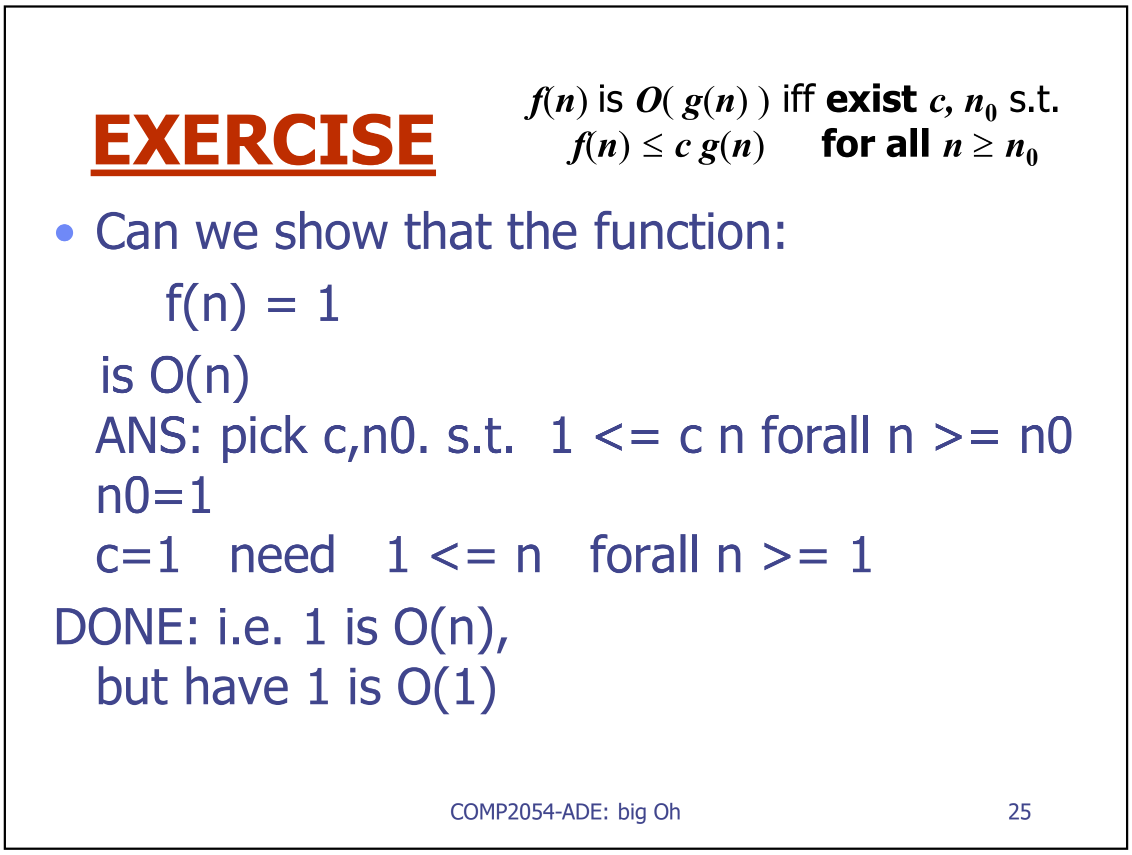
- Can we show that the function:

  f(n) = 1

  is O(n)

  ANS: (pause and try!)

TRY THIS FIRST



EXERCISE

$f(n)$ is $O(\,g(n)\,)$ iff **exist** c, n_0 s.t. $f(n) \le c\, g(n)$ **for all** $n \ge n_0$

- Can we show that the function:

  f(n) = 1

  is O(n)

  ANS: pick c,n0. s.t. 1 <= c n forall n >= n0

  n0=1

  c=1 need 1 <= n forall n >= 1

DONE: i.e. 1 is O(n),

but have 1 is O(1)

Firstly proceed in the standard fashion and substitute the given f and g to determine what we need to prove.
Need to show

Exists c, n0 s.t. 1 <= c . n for all n >= n0.

Picking c=1 is natural.

Then also can pick n0=1.

In this case the proof reduces to demonstrating

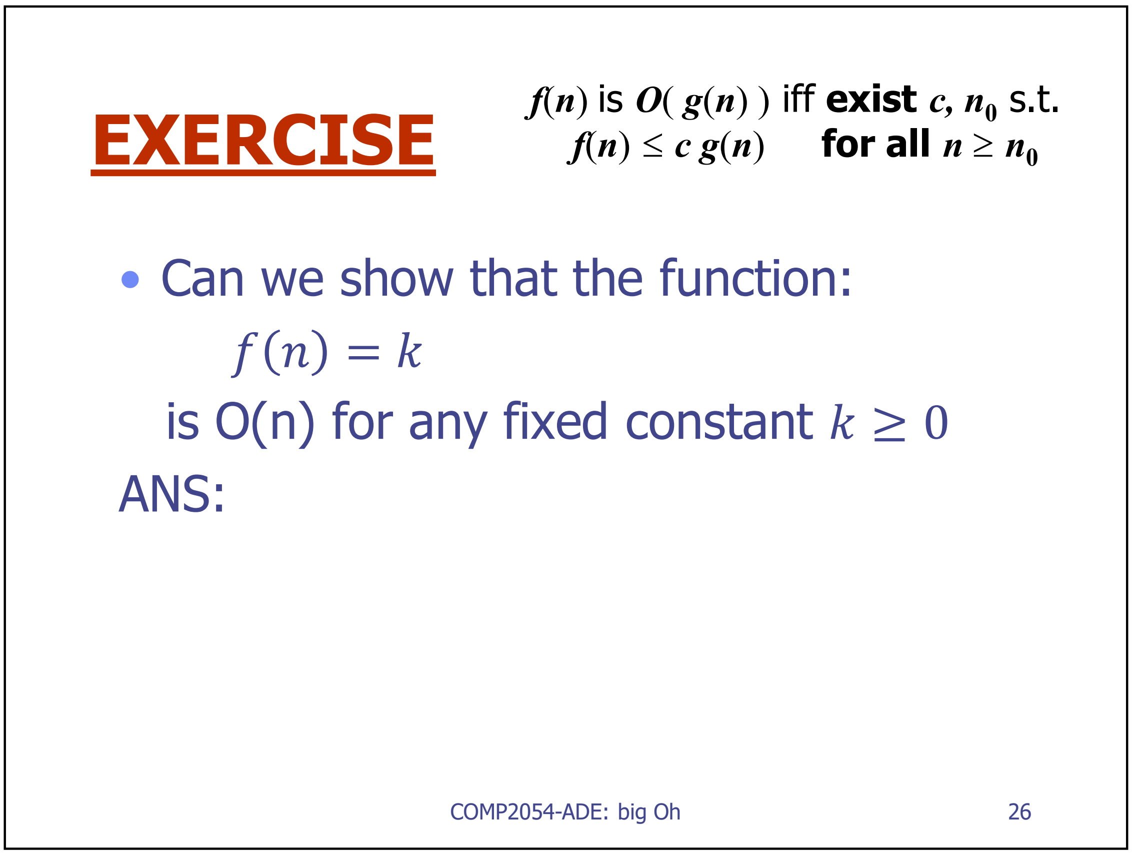
1 <= n for all n >= 1.

Which is trivially true.
Hence FROM THE DEFINITION we have 1 is O(n).

This is often surprising, but is because the every day usage of big-Oh also tends to include conventions of to pick the

Most informative option.
It is better to say 1 is O(1) as fewer functions are O(1) than are O(n).

This will be discussed again later.



EXERCISE

$f(n)$ is $O(g(n))$ iff **exist** c, n_0 s.t.
$f(n) \le c\, g(n)$ **for all** $n \ge n_0$

- Can we show that the function:

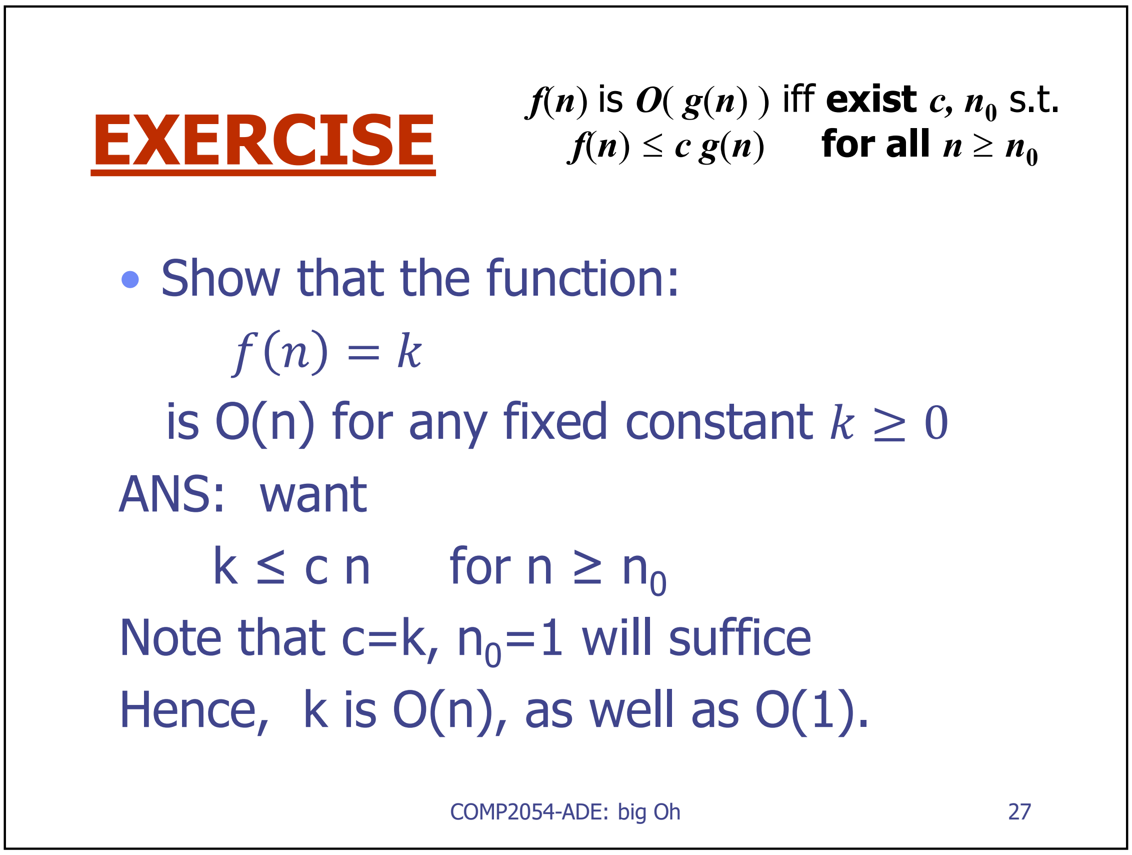
  $f(n) = k$

  is O(n) for any fixed constant $k \ge 0$

ANS:

COMP2054-ADE: big Oh 26

Again the proof is straightforward.



EXERCISE

$f(n)$ is $O(g(n))$ iff **exist** c, n_0 s.t.
$f(n) \le c\, g(n)$ **for all** $n \ge n_0$

- Show that the function:

$$f(n) = k$$

is O(n) for any fixed constant $k \ge 0$

ANS: want

$k \le c\, n$ for $n \ge n_0$

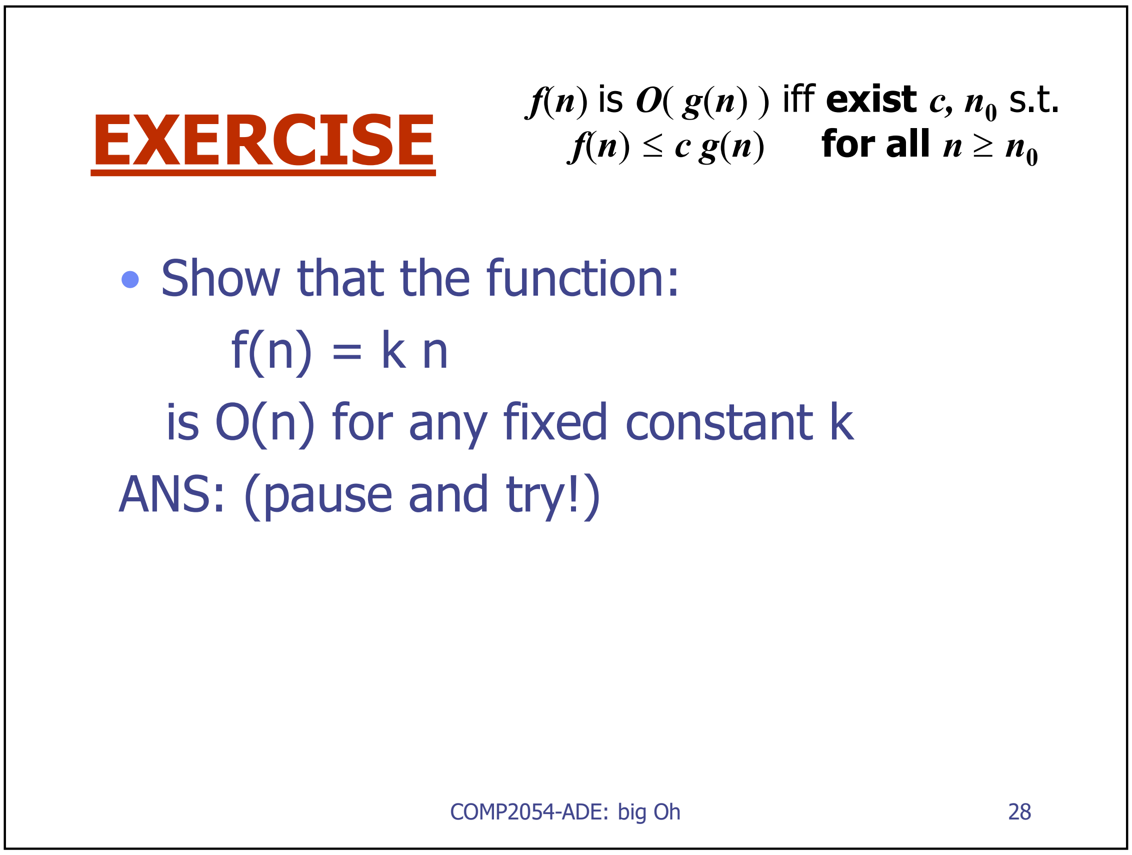
Note that $c=k$, $n_0=1$ will suffice

Hence, k is O(n), as well as O(1).

The structure is the same.

Note how the definitions of big-Oh mean that changing constants, like k, to other constant values, does not change the Big-Oh behaviour.

This is part of the point of the structure of the definition.



EXERCISE

$f(n)$ is $O(\,g(n)\,)$ iff **exist** c, n_0 s.t.
$f(n) \le c\, g(n)$ **for all** $n \ge n_0$

- Show that the function:

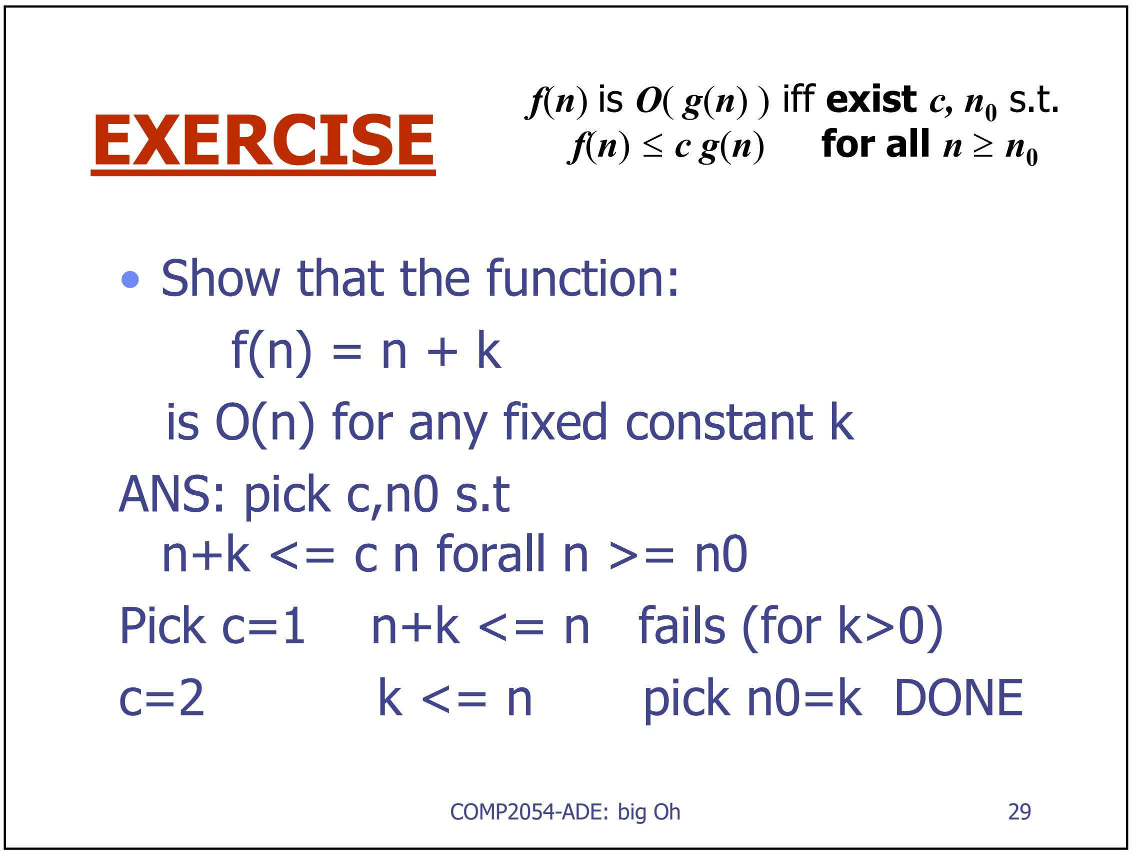
  f(n) = k n

  is O(n) for any fixed constant k

ANS: (pause and try!)

The proof is straightforward and left as an exercise.

Again not that it means changing the multiplier constant, does not affect the big-Oh.



EXERCISE

f(*n*) is *O*(*g*(*n*)) iff **exist** c, n_0 s.t. $f(n) \le c\, g(n)$ **for all** $n \ge n_0$

- Show that the function:

  f(n) = n + k

  is O(n) for any fixed constant k

ANS: pick c,n0 s.t
n+k <= c n forall n >= n0

Pick c=1 n+k <= n fails (for k>0)

c=2 k <= n pick n0=k DONE

A slightly more complex example.
Substituting f and g into the definition gives that we need

Exists c,n0. s.t. n+k <= c n forall n >= n0

ie. Exists c,n0. s.t. forall n >= n0. n+k <= c n

We can try c=1 and then find it fails when k>0.

It would be very wrong to conclude that this shows it is not O(n).

The definition says "Exists c" not "forall c".
Showing that one specific choice of c fails does not mean that all values will fail.
E.g. in this case

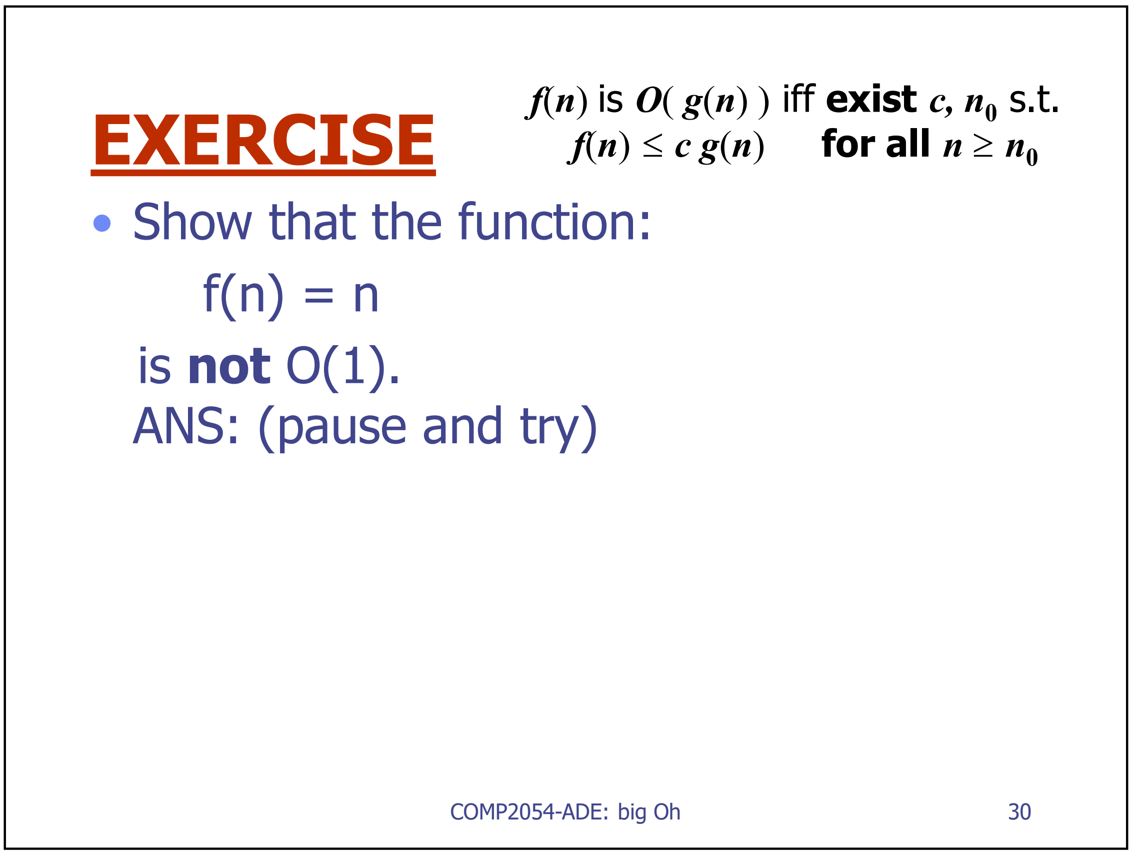
Pick c=2, then means we have to show

Exists n0. s.t. forall n >= n0. n+k <= 2 n

Exists n0. s.t. forall n >= n0. k <= n

Which is trivially satisfied by n0=k.

Other choices for c will also work.



EXERCISE

$f(n)$ is $O(g(n))$ iff **exist** c, n_0 s.t. $f(n) \leq c\, g(n)$ **for all** $n \geq n_0$

- Show that the function:

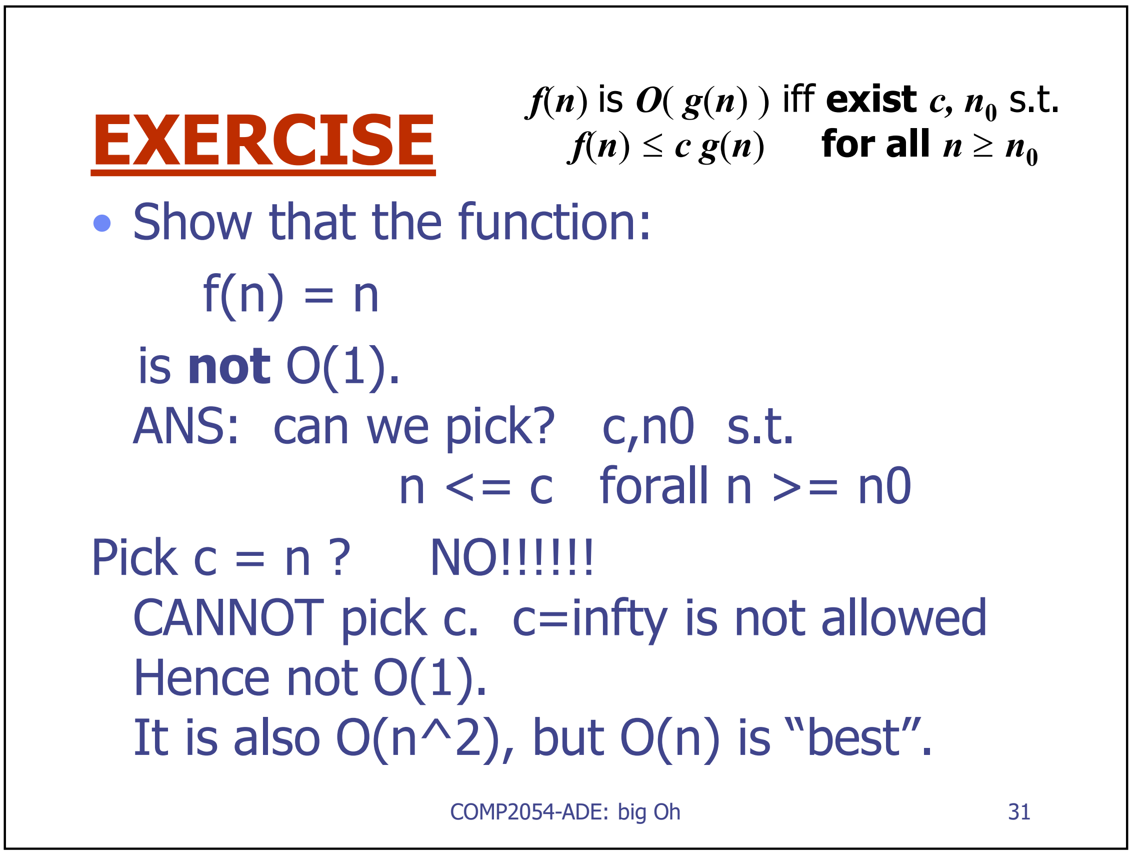
  f(n) = n

  is **not** O(1).

  ANS: (pause and try)

COMP2054-ADE: big Oh 30

TRY THIS FIRST IF OFFLINE



EXERCISE

$f(n)$ is $O(g(n))$ iff **exist** c, n_0 s.t.
$f(n) \leq c\, g(n)$ **for all** $n \geq n_0$

- Show that the function:

  f(n) = n

  is **not** O(1).

  ANS: can we pick? c,n0 s.t.

  n <= c forall n >= n0

Pick c = n ? NO!!!!!!

CANNOT pick c. c=infty is not allowed

Hence not O(1).

It is also O(n^2), but O(n) is "best".

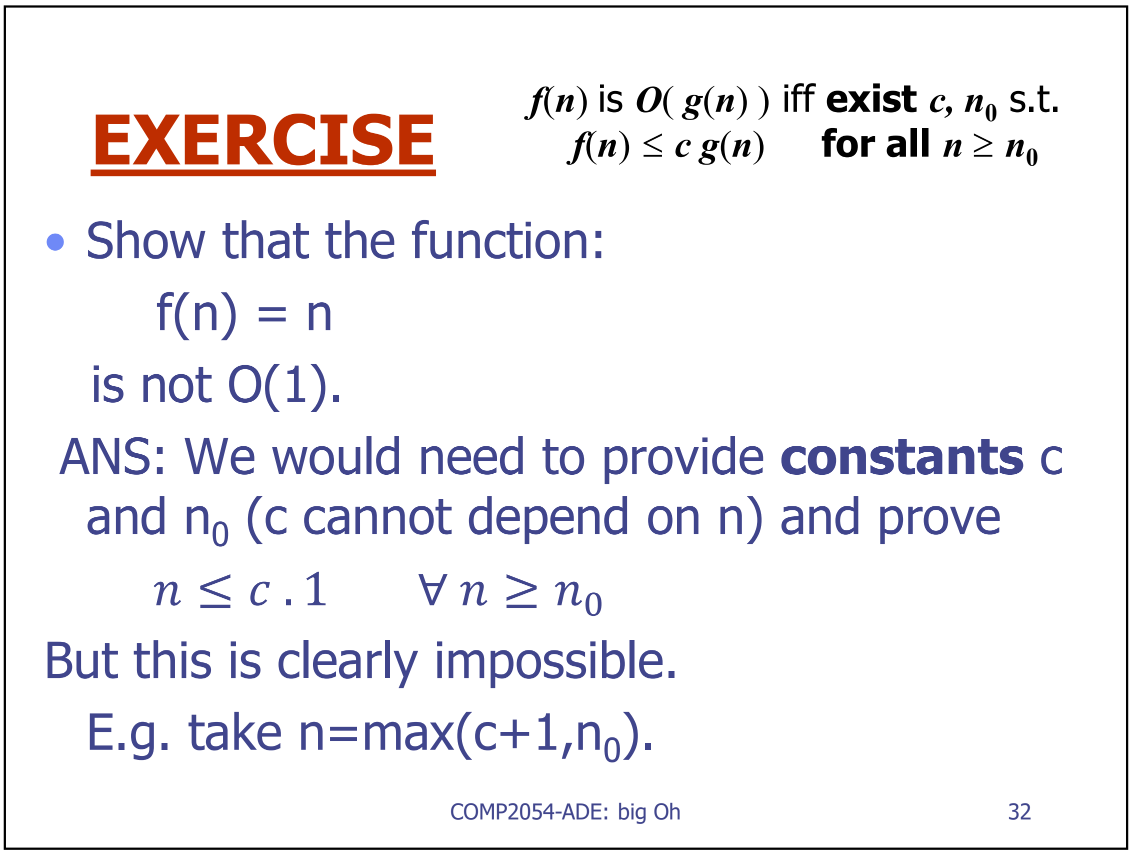
Note that as before the first step is just to substitute the given data into the definition in order to see what we would need to prove.

We need
Exists c,n0. forall n >= n0. n <= c

But this is impossible for any finite fixed choice of c.
Note that it is implicit in the definitions that c is a real number, and no real number is infinite.

We can always add 1 to any number and get a larger number.
Numbers are arbitrarily large, but this is very different from saying that some number is infinite.

This is important because it shows that using the big-Oh definition does convey useful information.



EXERCISE

$f(n)$ is $O(\ g(n)\ )$ iff **exist** $c,\ n_0$ s.t.
$f(n) \leq c\ g(n)$ **for all** $n \geq n_0$

- Show that the function:

  f(n) = n

  is not O(1).

ANS: We would need to provide **constants** c and n_0 (c cannot depend on n) and prove

$$n \leq c\,.\,1 \qquad \forall\, n \geq n_0$$

But this is clearly impossible.

E.g. take $n=\max(c+1,n_0)$.

This slide is just the same argument as the previous slide, but written out in a slightly more formal fashion.

BREAK until next lecture lec02b