Big-Oh - part I

CMPT-225

Recall: If f, g are functions $f: \mathbb{N} \to \mathbb{N}$, $g: \mathbb{N} \to \mathbb{N}$

f is $O(g)$ means there are constants $n_0, c > 0$

s.t. for every $n > n_0$, $f(n) \leq c \cdot g(n)$.

that is,

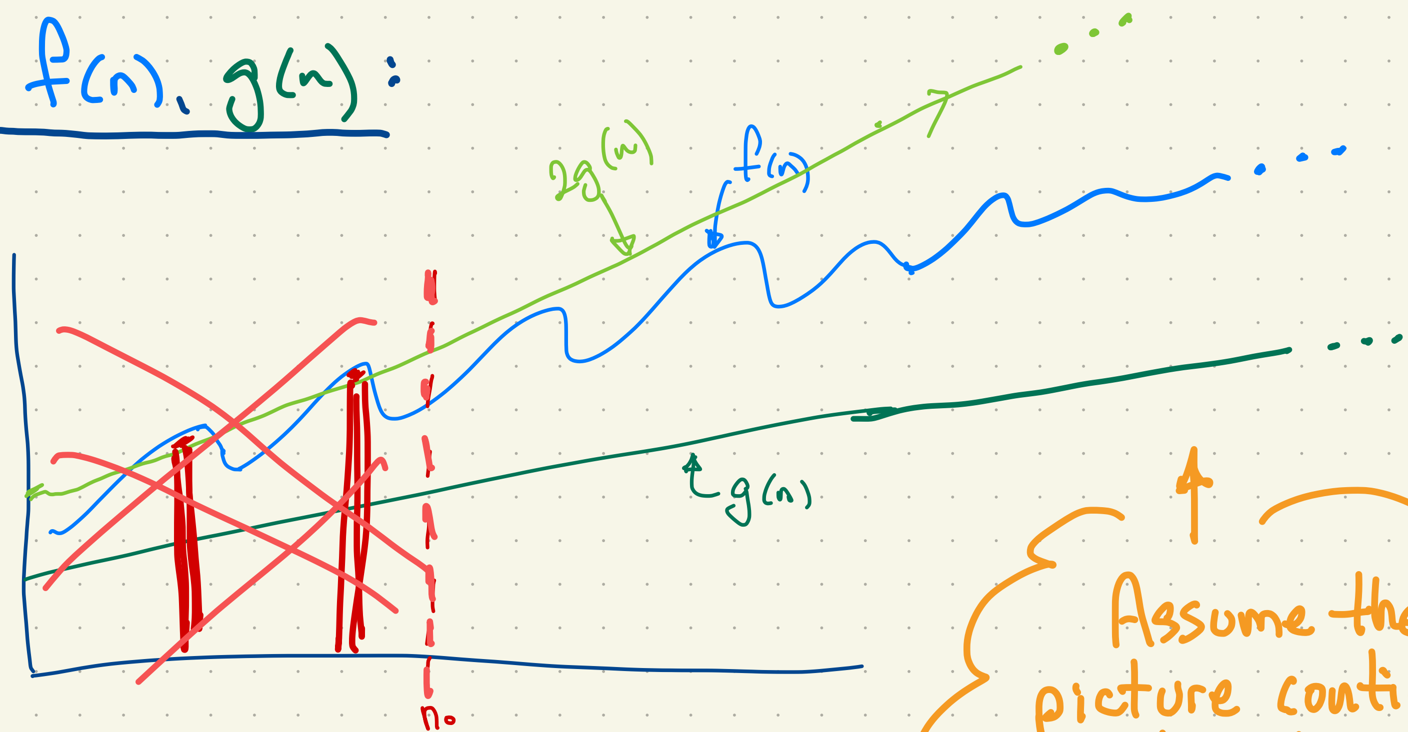
for all but finitely many "small" values of n,

$$f(n) \leq c\, g(n)$$

or f grows no faster than g (asymptotically)

we typically write $f(n) = O(g(n))$

Consider $f(n)$, $g(n)$:



Assume the picture continues to look like this as $n \to \infty$

Claim: $f(n) = O(g(n))$

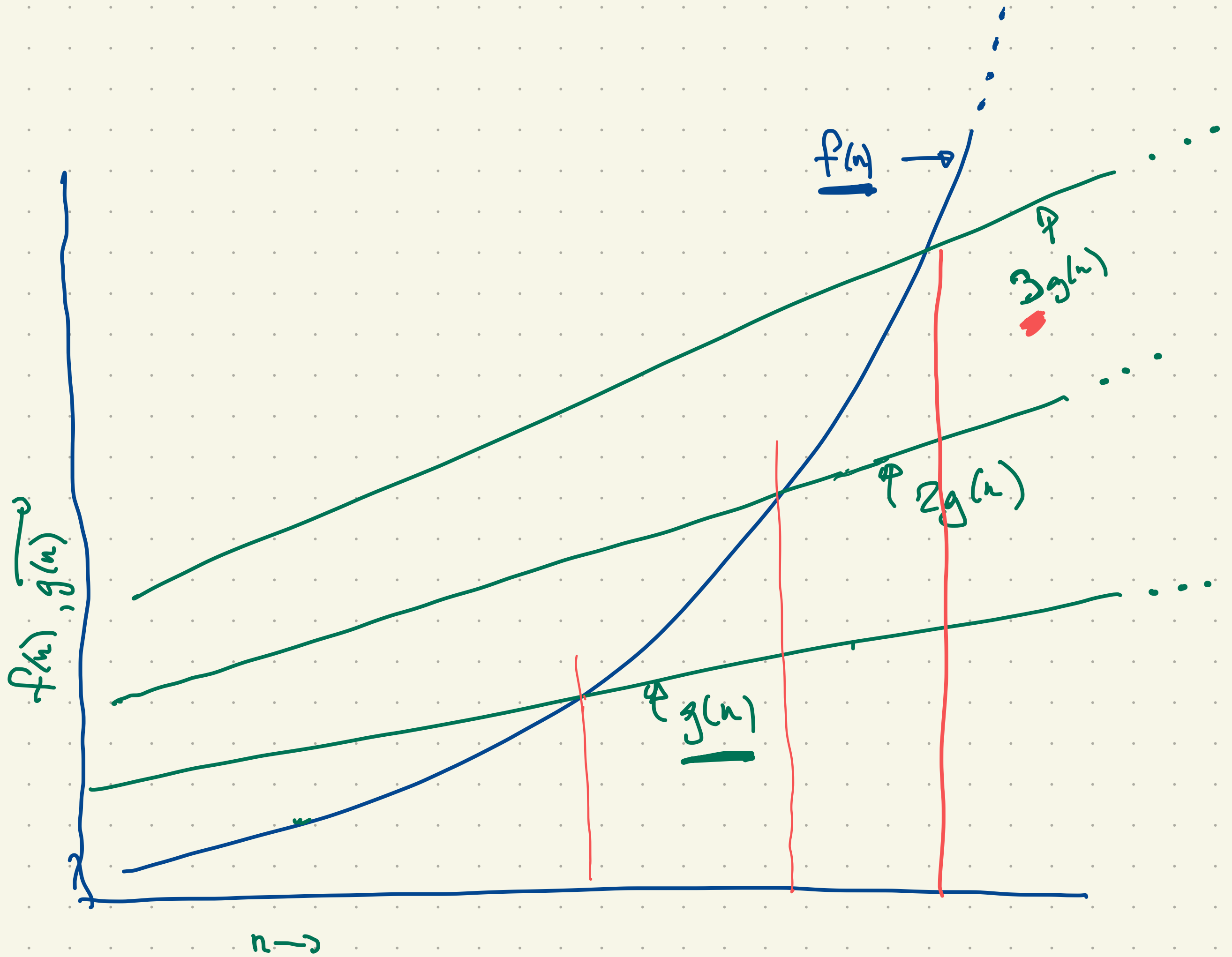
Why - when $f(n) > g(n)$ for all n?

- but $f(n) < 2 \cdot g(n)$, for almost all values of n.
- choose n_0 to "exclude" the red values of n

- Now: $f(n) < 2 \cdot g(n)$ for all $n > n_0$.

  So: $f(n) = O(g(n))$

Also:



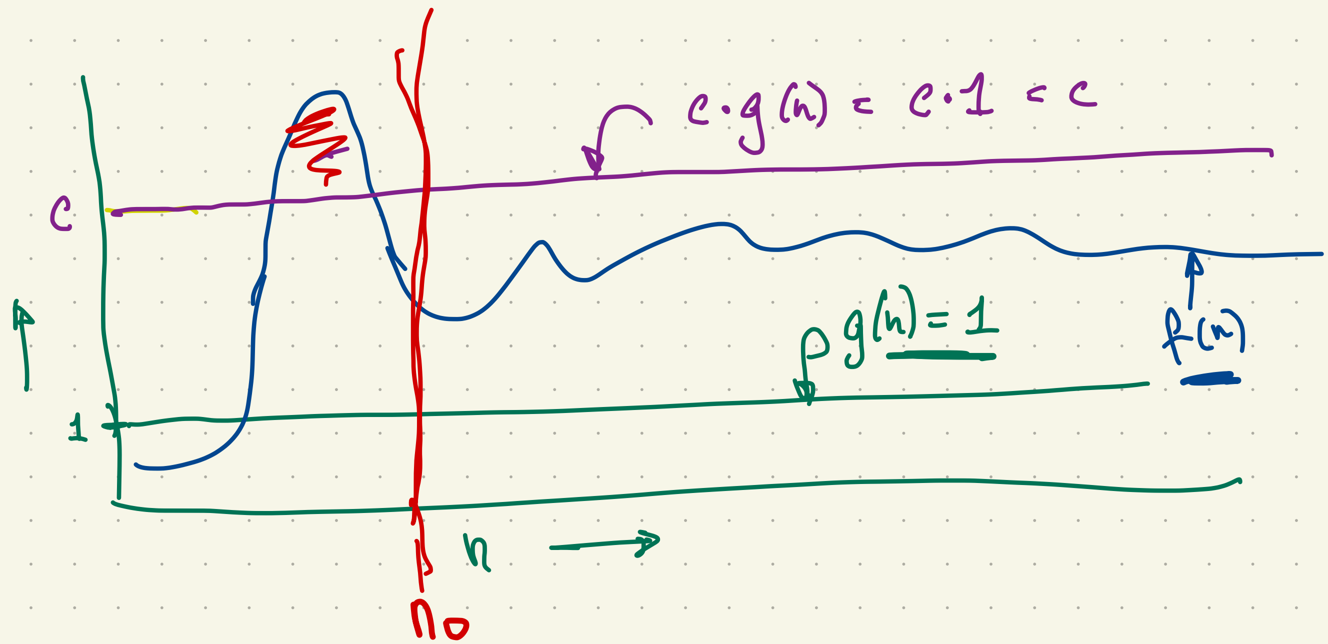
claim: $f(n)$ is <u>not</u> $O(g(n))$

However large we choose c, n_0, there will be some k (larger than n_0) s.t.:

$$n > k \Rightarrow f(n) > c \cdot g(n)$$

Consider: $O(1)$ (i.e. $g(n) = 1$)

$f(n) = O(1)$ iff $\exists\, n_0, c > 0$ s.t. $\forall n > n_0$ $f(n) \leq c \cdot 1$



- for every $n > n_0$, $f(n) < c$
- so, $f(n)$ grows no faster than a constant
- so $f(n)$ is asymptotically bounded by a constant.

The constant does not matter:

Fact: $f(n) = O(1)$ iff $f(n) = O(10^{27})$ iff $f(n) = O(1/10^{27})$

Suppose $f(n) = O(10^{27})$ $(*)$

Claim: $f(n)$ is also $O(1/10^{27})$

$(*)$ means $\exists n_0, c > 0$ s.t $n > n_0 \Rightarrow f(n) \leq c \cdot 10^{27}$

Want to show: $\exists n_0, c' > 0$ s.t. $n > n_0 \Rightarrow f(n) \leq c' \cdot \frac{1}{10^{27}}$

Choose c' big enough that $c' \cdot \frac{1}{10^{27}} \geq c \cdot 10^{27}$

Eg: $c' = c \cdot 10^{54}$

Then: for all $n > n_0$, $f(n) \leq c \cdot 10^{27}$

$$\leq c \cdot 10^{-27} \cdot 10^{54}$$

$$\leq c' \cdot 10^{-27}$$

$(54 - 27 = 27)$

So: $f(n) = O(10^{-27})$

Asymptotic Notation (e.g. Big-Oh)

- Is not "about" algorithms
- Is a tool for describing (growth of) functions
- It is useful for describing functions related to algorithms + data structures

  e.g. – minimum or maximum time taken
  – minimum or maximum space needed
  ⋮

- We use it so often for worst-case time for an algorithm that we often leave implicit a statement like "let $T(n)$ be the max. time taken by algorithm A on an input of size at most n."

  This statement is essential.

Ex. Complexity of Palindrome Checking.

- using a stack & queue
- Algorithm: 1) insert all tokens into a stack & a queue
  2) repeat: · pop one token; deque one token
  · if different, report 'no'.

- size of input = number of symbols or tokens
- each token is:
  - pushed on the stack, $O(1)$
  - enqueued on the queue, $O(1)$
  - popped off the stack, $O(1)$
  - dequeued from the queue, $O(1)$
  - compared to one other token. $O(1)$

  All together $O(1)$

- n tokens $\Longrightarrow$ n times $O(1)$ time in total
- So: $T(n) = \underbrace{n \cdot O(1) = O(n)}_{?}$.

What does $n \cdot O(1) = O(n)$ mean?

It means : $f(n) = O(1)$ iff $n \cdot f(n) = O(n)$.

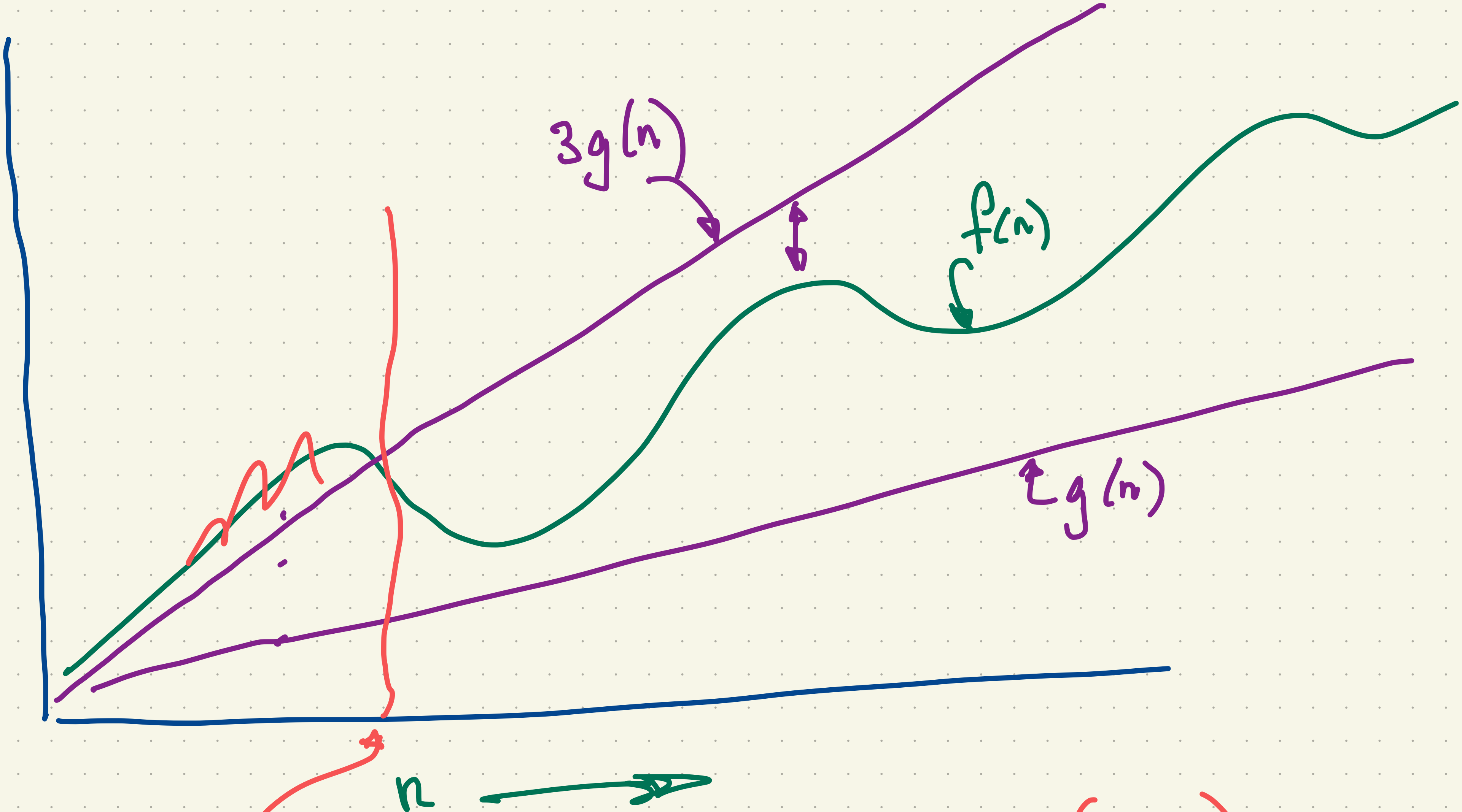
To see it is true:

$$f(n) = O(1) \iff \exists c > 0 \text{ s.t. } f(n) < c, \text{ for any } n \in \mathbb{N}$$

$$\iff \exists c > 0 \text{ s.t. } n \cdot f(n) < c \cdot n, \text{ for any } n \in \mathbb{N}$$

$$\iff n \cdot f(n) = O(n)$$

END



Claim: $f(n) = O(g(n))$

$f(n) = \underline{O(n)}$ means (ie. $g(n) = n$)

$\exists\, n_0, c \geq 0$ s.t. $\forall n > n_0$ $f(n) < c \cdot n$