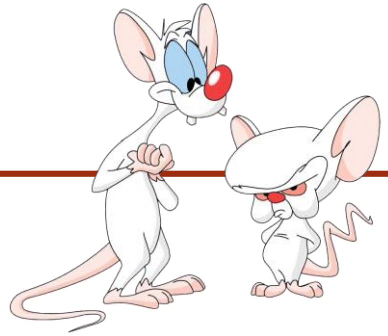
COMP 250

INTRODUCTION TO COMPUTER SCIENCE

Week 6-2 : Asymptotic Notation 1

Giulia Alberini, Fall 2020

WHAT ARE WE GOING TO DO IN THIS VIDEO?



- Analysis of algorithms
- Asymptotic notation
  - Big-Oh, $O(\cdot)$
- Coming next
  - Big-Omega, $\Omega(\cdot)$
  - Big-Theta, $\Theta(\cdot)$

ANALYSIS OF ALGORITHMS

- Often we are interested in knowing how much time an algorithm needs to perform a given task.

- Typically, the time taken by an algorithm depends on the input and it grows with the size of such input. This is why we usually describe the running time of an algorithm with a *function* of the size of its input.

- What do we mean by "size of input"?
  What do we mean by "running time"?

SIZE OF INPUT, RUNNING TIME

- The notion of *input size* depends on the problem being studied, and it can therefore vary depending on the algorithm analyzed
  - It can be the number of elements in the input (e.g. the length of an array in a sorting algorithm)
  - It can be the number of bits required to represent the input (e.g. when multiplying two numbers)
  - It can be described by multiple numbers rather that one (e.g. algorithms that work with graphs)

- The running time of an algorithm is the number of primitive operations (e.g. evaluating an expression, assigning a value, returning from a method,…) executed.

Where t_i is the number of times the condition of the while loop is checked for the specific i

EXAMPLE – INSERTION SORT

```
insertionSort(list) {
   for i from 1 to n-1 {
      element = list[i]
      k = i
      while(k>0 && element<list[k-1]){
         list[k] = list[k-1]
         k--
      }
      list[k] = element
   }
}
```


Cost	Times
c_1	n
c_2	$n-1$
c_3	$n-1$
c_4	$\sum_{i=1}^{n-1} t_i$
c_5	$\sum_{i=1}^{n-1}(t_i-1)$
c_6	$\sum_{i=1}^{n-1}(t_i-1)$
c_7	$n-1$

EXAMPLE – INSERTION SORT

Cost	Times
c_1	n
c_2	$n-1$
c_3	$n-1$
c_4	$\sum_{i=1}^{n-1} t_i$
c_5	$\sum_{i=1}^{n-1}(t_i - 1)$
c_6	$\sum_{i=1}^{n-1}(t_i - 1)$
c_7	$n-1$

So, we can represent the running time of insertion sort as a function of the size of its input as follows:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=1}^{n-1} t_i + c_5 \sum_{i=1}^{n-1}(t_i - 1) + c_6 \sum_{i=1}^{n-1}(t_i - 1) + c_7(n-1)$$

Even for inputs of the same size, the running time might be different.

EXAMPLE – INSERTION SORT BEST CASE

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=1}^{n-1} t_i + c_5 \sum_{i=1}^{n-1} (t_i - 1) + c_6 \sum_{i=1}^{n-1} (t_i - 1) + c_7(n-1)$$

The best case occurs when the array is already sorted. In this case, the condition of the while will be checked only once. That is, $t_i = 1$ for all i. Therefore,

$$T_{best}(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1)$$
$$= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$$

Which we can express as $T_{best}(n) = an + b$, for some constants a and b.

EXAMPLE – INSERTION SORT WORST CASE

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=1}^{n-1} t_i + c_5 \sum_{i=1}^{n-1} (t_i - 1) + c_6 \sum_{i=1}^{n-1} (t_i - 1) + c_7(n-1)$$

The worst case occurs when the array is already sorted, but in the reverse order. In this case, $t_i = i + 1$ for all i.

Note that

$$\sum_{i=1}^{n-1} (i+1) = \sum_{i=2}^{n} i = \frac{1}{2} n(n+1) - 1$$

$$\sum_{i=1}^{n-1} i = \frac{1}{2}(n-1)n$$

EXAMPLE – INSERTION SORT WORST CASE

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=1}^{n-1} t_i + c_5 \sum_{i=1}^{n-1} (t_i - 1) + c_6 \sum_{i=1}^{n-1} (t_i - 1) + c_7(n-1)$$

The worst case occurs when the array is already sorted, but in the reverse order. In this case, $t_i = i + 1$ for all i. Therefore,

$$T_{worst}(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{1}{2}n(n+1) - 1\right) + (c_5 + c_6)\left(\frac{1}{2}n(n-1)\right) + c_7(n-1)$$

$$= \frac{1}{2}(c_4 + c_5 + c_6)n^2 + \left(c_1 + c_2 + c_3 + c_7 + \frac{1}{2}(c_4 + c_5 + c_6)\right)n - (c_2 + c_4 + c_5 + c_8)$$

Which we can express as $T_{worst}(n) = an^2 + bn + c$, for some constants a, b, and c.

"BIG-PICTURE" APPROACH

- When we analyze algorithms we use what is referred to as "the big-picture" approach. What we care about is the growth rate of the running time. We look at how the running time increases with the size of the input in the limit, as the size of the input increases without bound.

- To perform this analysis:
  - Find the running time as a function of the input size
  - Use asymptotic notation to express this function

ASYMPTOTIC NOTATION

- Asymptotic notations apply to functions.

- We will use asymptotic notations to describe the running time of algorithms. This means that the function to which we apply the asymptotic notation describes the running time of algorithms.

- In general, asymptotic notation can be applied to functions that describe other characteristics of an algorithm, or functions that have nothing to do with algorithms.

TOWARDS A FORMAL DEFINITION OF BIG OH

Let $\mathrm{T}(n)$ be a function that describes the time it takes for some algorithm to terminate on input size n.

We would like to express how $\mathrm{T}(n)$ grows with n, as n becomes large i.e. *asymptotic* behavior.

Unlike with limits, we want to say that $\mathrm{T}(n)$ grows like certain *simpler* functions such as $\boldsymbol{log_2 n,\ \ n,\ \ n^2, \dots, 2^n}$, etc.

PRELIMINARY (INCOMPLETE) FORMAL DEFINITION

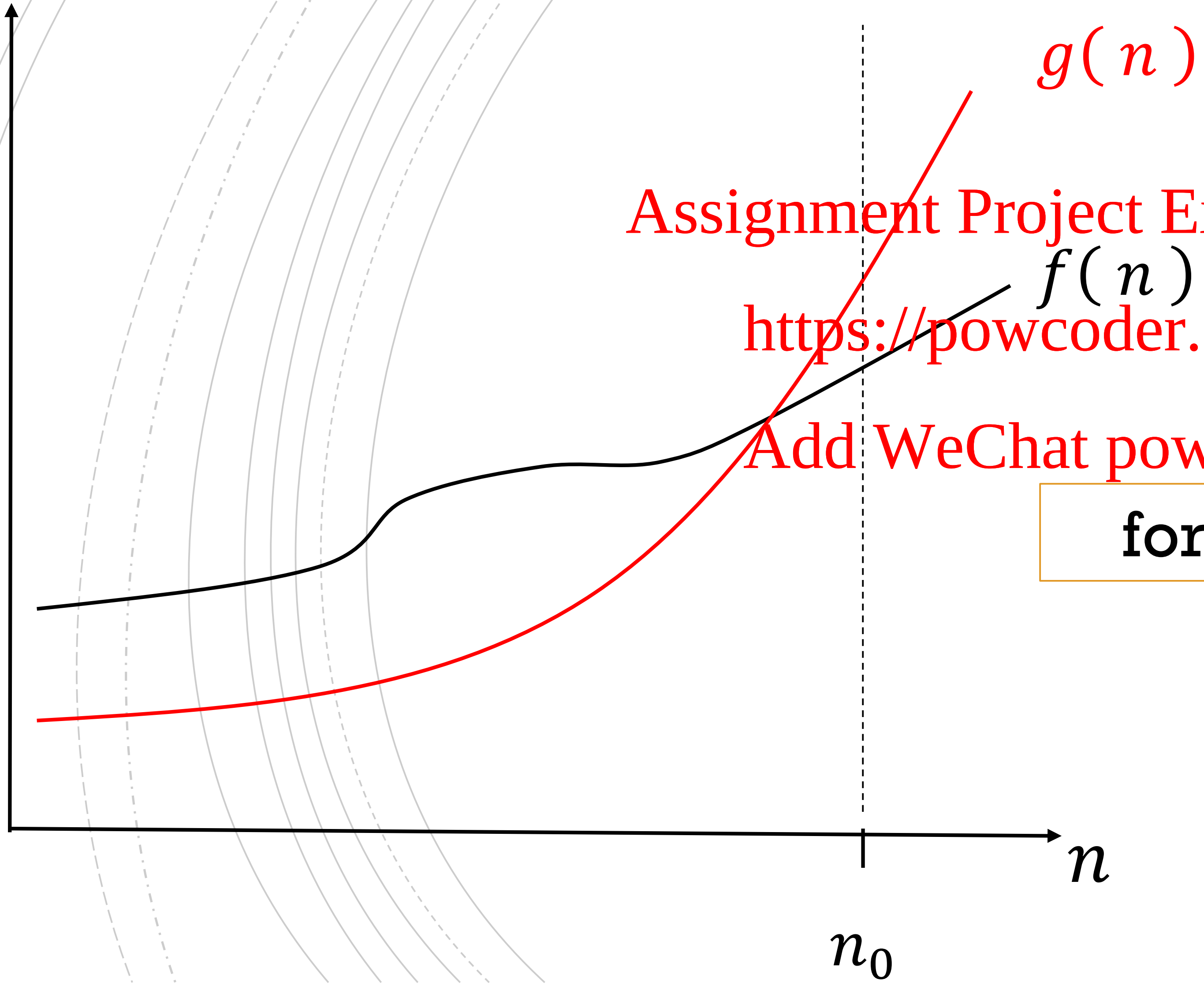
Let $f(n)$ and $g(n)$ be two functions, where $n \geq 0$.

We say that $f(n)$ is *asymptotically bounded above* by $g(n)$ if there exists n_0 such that, for all $n \geq n_0$,

$$f(n) \leq g(n).$$

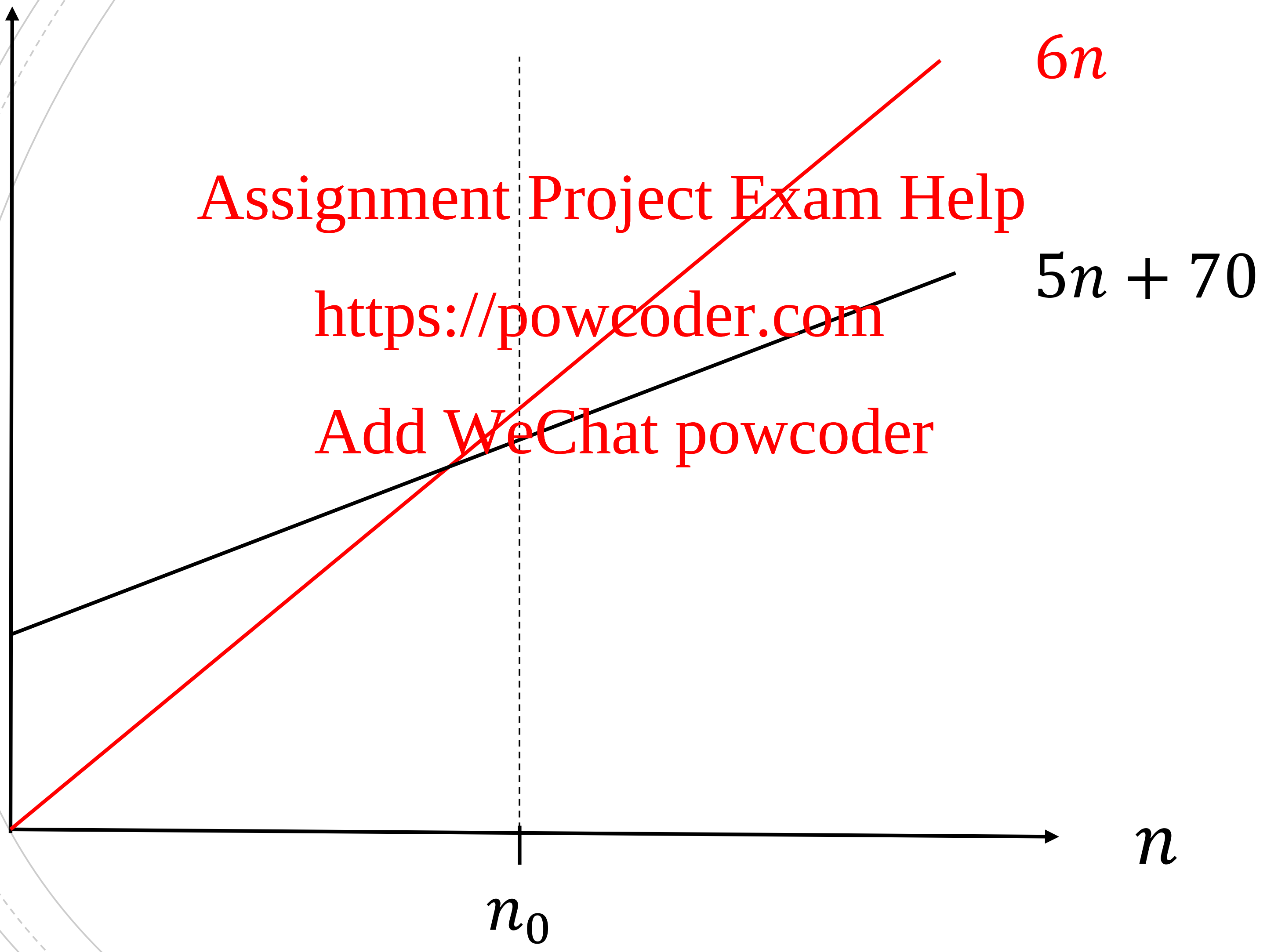
This is not yet a formal definition *of big O.*

GRAPHICALLY



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

EXAMPLE



EXAMPLE – PROOF

Claim: $5n + 70$ is *asymptotically bounded above* by $6n$.

To prove: show that there exists an n_0 such that, for all $n \geq n_0$,

$$5n + 70 \leq 6n$$

EXAMPLE – PROOF

Claim: $5n + 70$ is *asymptotically bounded above* by $6n$.

To prove: show that there exists an n_0 such that, for all $n \geq n_0$,

$$5n + 70 \leq 6n$$

Proof: Note that,

$$5n + 70 \leq 6n \iff 70 \leq n$$

"$\iff$" means "if and only if"
i.e. logical equivalence

EXAMPLE – PROOF

Claim: $5n + 70$ is *asymptotically bounded above* by $6n$.

To prove: show that there exists an n_0 such that, for all $n \geq n_0$,

$$5n + 70 \leq 6n$$

Proof: Note that,

$$5n + 70 \leq 6n \iff 70 \leq n$$

Thus, we can use $n_0 = 70$.

TOWARDS A FORMAL DEFINITION OF BIG OH

Let $\mathrm{T}(n)$ be a function that describes the time it takes for some algorithm on input size n.

We would like to express how $\mathrm{T}(n)$ grows with n, as n becomes large i.e. *asymptotic* behavior.

Unlike with limits, we want to say that $\mathrm{T}(n)$ grows like certain *simpler* functions such as $\boldsymbol{log_2 n,\ \ n,\ \ n^2, \dots, 2^n}$, etc.

FORMAL DEFINITION OF BIG O

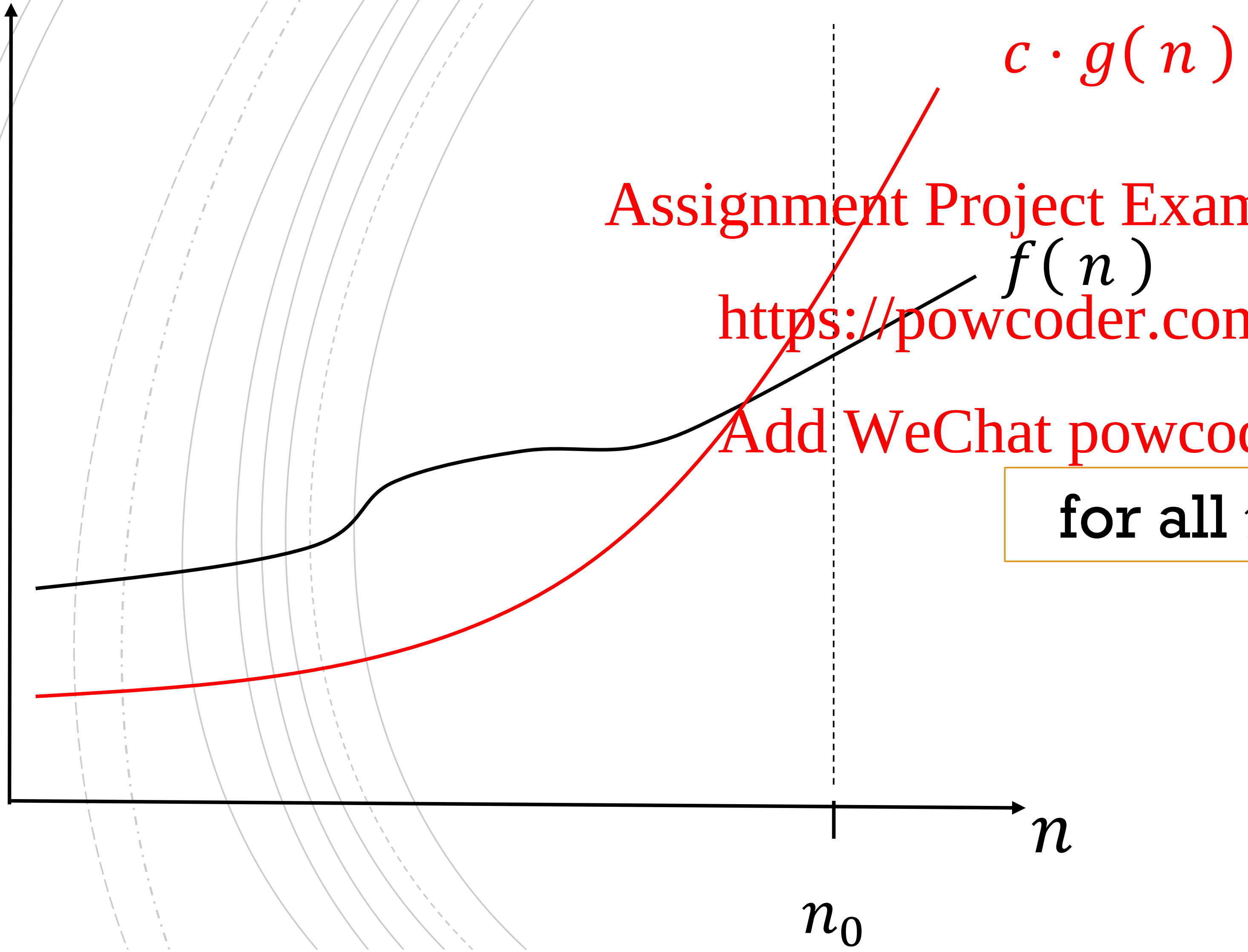
Given a function $g(n)$, we denote by $O(g(n))$ ("big-oh of g of n") the following set of functions

$$O\big(g(n)\big) = \{f(n)\text{: there exist positive constants } c \text{ and } n_0 \text{ such that}$$

$$f(n) \le cg(n) \text{ for all } n \ge n_0\ \}$$

We use the O-notation to describe an **asymptotic upper bound.**

GRAPHICALLY



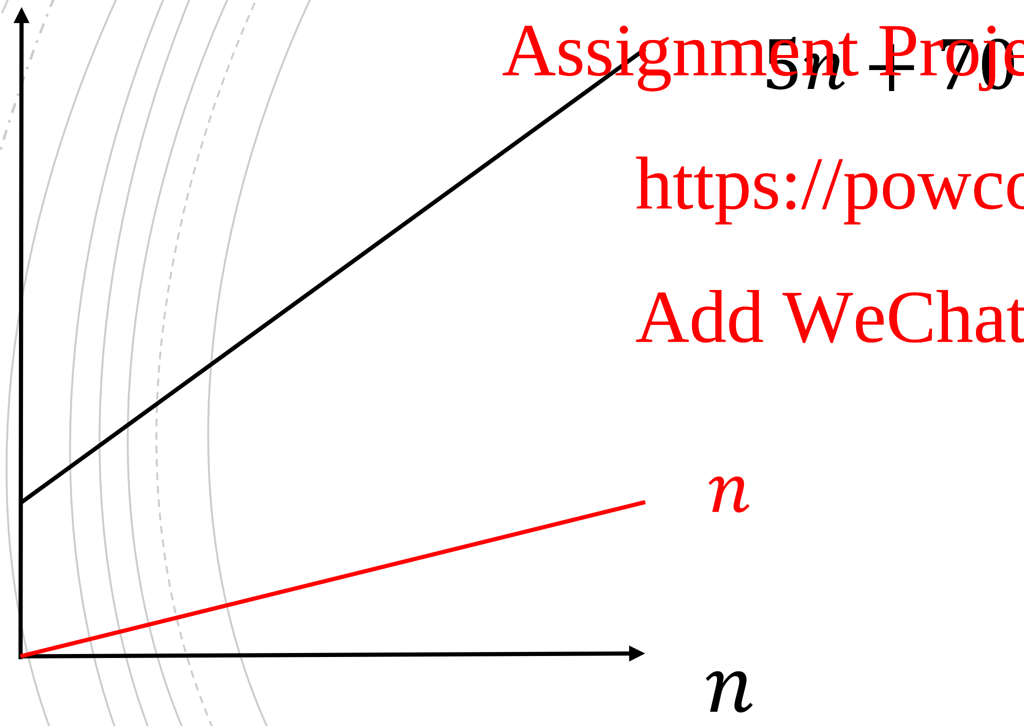
for all $n_0 \geq n, \quad f(n) \leq c \cdot g(n)$

OBSERVATIONS

- Note that we sometime write $f(n) = O(g(n))$ (and say "$f(n)$ is $O(g(n))$") to indicate that the function $f(n)$ is a member of the set $O\big(g(n)\big)$. (i.e. $f(n) \in O(g(n))$)

- Moreover, we sometimes find O-notation used to describe asymptotically tight bounds, but the O-notation by definition only claims **asymptotic upper bound**.

EXAMPLE



Claim: $5n + 70$ is $O(n)$.

EXAMPLE – PROOF

Claim: $5n + 70$ is $O(n)$.

To prove: show that there exists a c and an n_0 such that, for all $n \geq n_0$,

$$5n + 70 \leq c \cdot n$$

EXAMPLE – PROOF

Claim: $5n + 70$ is $O(n)$.

To prove: show that there exists a c and an n_0 such that, for all $n \geq n_0$,

$$5n + 70 \leq c \cdot n$$

Proof:

$$5n + 70 \leq ?$$

EXAMPLE – PROOF

Claim: $5n + 70$ is $O(n)$.

To prove: show that there exists a c and an n_0 such that, for all $n \geq n_0$,

$$5n + 70 \leq c \cdot n$$

Proof 1:

$$5n + 70 \leq 5n + 70n, \qquad \text{if } n \geq 1$$

EXAMPLE – PROOF

Claim: $5n + 70$ is $O(n)$.

To prove: show that there exists a c and an n_0 such that, for all $n \geq n_0$,

$$5n + 70 \leq c \cdot n$$

Proof 1:

$$5n + 70 \leq 5n + 70n, \qquad \text{if } n \geq \mathbf{1}$$

$$= \mathbf{75}n$$

So we can pick $\boldsymbol{c = 75}$ and $\boldsymbol{n_0 = 1}$

EXAMPLE – PROOF

Claim: $5n + 70$ is $O(n)$.

To prove: show that there exists a c and an n_0 such that, for all $n \geq n_0$,

$$5n + 70 \leq c \cdot n$$

Proof 2:

$$5n + 70 \leq 5n + 6n, \qquad \text{if } n \geq 12$$

EXAMPLE – PROOF

Claim: $5n + 70$ is $O(n)$.

To prove: show that there exists a c and an n_0 such that, for all $n \geq n_0$,

$$5n + 70 \leq c \cdot n$$

Proof 2:

$$5n + 70 \leq 5n + 6n, \qquad \text{if } n \geq \mathbf{12}$$

$$= \mathbf{11}n$$

So we can pick $\boldsymbol{c = 11}$ and $\boldsymbol{n_0 = 12}$

EXAMPLE – PROOF

Claim: $5n + 70$ is $O(n)$.

To prove: show that there exists a c and an n_0 such that, for all $n \geq n_0$,

$$5n + 70 \leq c \cdot n$$

Proof 3:

$$5n + 70 \leq 5n + n, \qquad \text{if } n \geq 70$$

EXAMPLE – PROOF

Claim: $5\,n + 70$ is $O(n)$.

To prove: show that there exists a c and an n_0 such that, for all $n \geq n_0$,

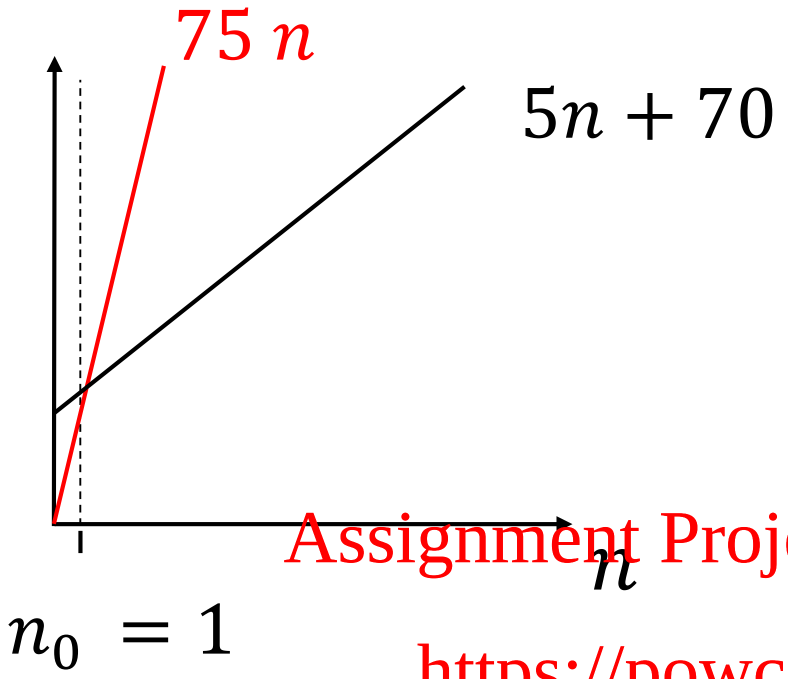
$$5n + 70 \leq c \cdot n$$

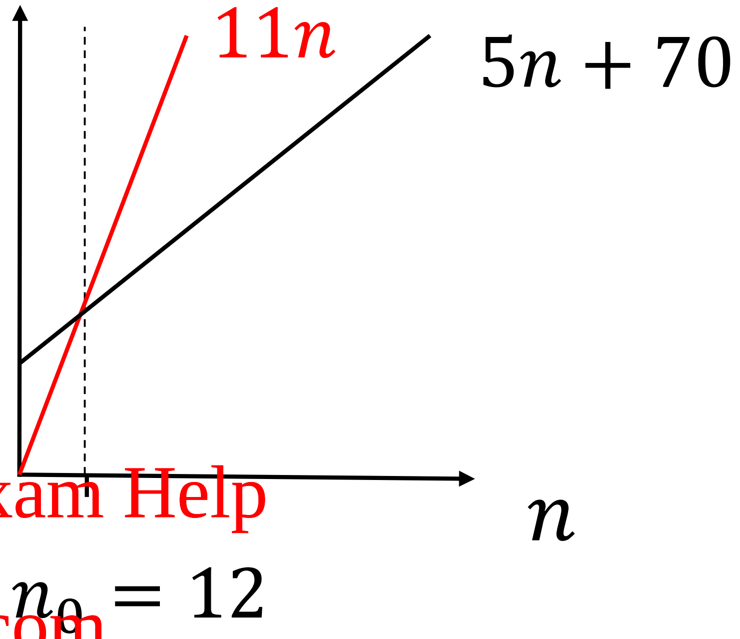
Proof 3:

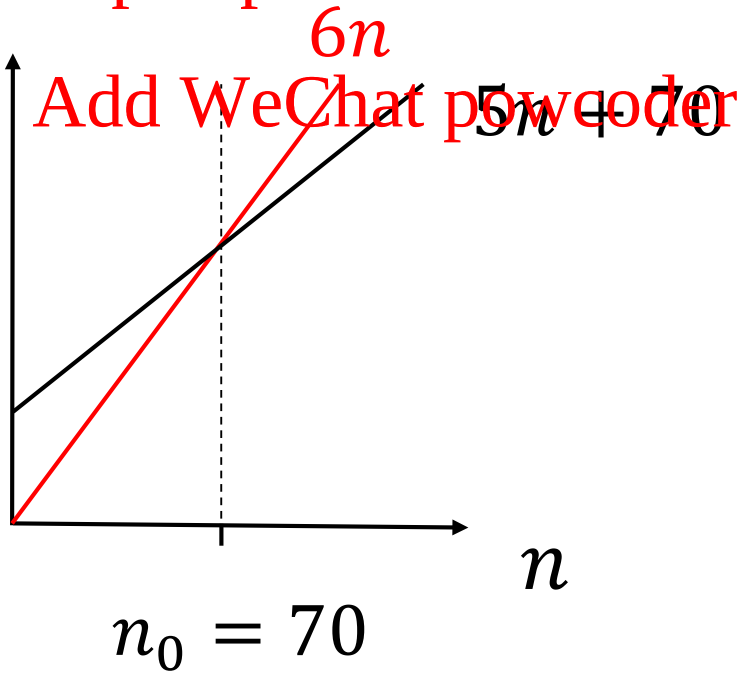
$$5\,n + 70 \leq 5\,n + n, \qquad \text{if } n \geq \mathbf{70}$$

$$= \mathbf{6}n$$

So we can pick $\boldsymbol{c} = \mathbf{6}$ and $\boldsymbol{n_0} = \mathbf{70}$


$75\,n$
$5n + 70$
n
$n_0 = 1$


$11n$
$5n + 70$
n
$n_0 = 12$


$6n$
$5n + 70$
n
$n_0 = 70$

EXAMPLE – INCORRECT PROOF

Claim: $5n + 70$ is $O(n)$.

Incorrect Proof:

$$
\begin{aligned}
5n + 70 &\leq cn \\
5n + 70n &\leq cn, \quad n \geq 1 \\
75n &\leq cn
\end{aligned}
$$

$$\text{Thus,} \quad c > 75, \quad n_0 = 1 \quad \text{works.}$$

Q: Why is this incorrect?

EXAMPLE – INCORRECT PROOF

Claim: $5n + 70$ is $O(n)$.

Incorrect Proof:

$$\begin{aligned} 5n + 70 &\leq cn \\ 5n + 70n &\leq cn, \quad n \geq 1 \\ 75n &\leq cn \end{aligned}$$

Thus, $c > 75$, $n_0 = 1$ works.

Q: Why is this incorrect?
A: Because we don't know which line follows logically from which.

EXAMPLE 2

Claim: $8\,n^2 \; - 17n + 46$ **is** $O(n^2)$.

Proof 1: $8\,n^2 \; - 17n + 46$

EXAMPLE 2

Claim: $8\,n^2 \; - 17n + 46$ is $O(n^2)$.

Proof 1: $8\,n^2 \; - 17n + 46$

$\leq \; 8\,n^2 + 46\,n^2, \;$ for $n \geq \; 1$

EXAMPLE 2

Claim: $8\,n^2 - 17n + 46$ is $O(n^2)$.

Proof 1: $8\,n^2 - 17n + 46$

$\leq 8\,n^2 + 46\,n^2$, for $n \geq \mathbf{1}$

$\leq \mathbf{54}\,n^2$

So we can take $\boldsymbol{c} = \mathbf{54}$ and $\boldsymbol{n_0} = \mathbf{1}$

EXAMPLE 2

Claim: $8\,n^2 - 17n + 46$ **is** $O(n^2)$.

Proof 2: $8\,n^2 - 17n + 46$

EXAMPLE 2

Claim: $8\,n^2 \;-17n + 46$ is $O(n^2)$.

Proof 2: $8\,n^2 \;- 17n + 46$

$\leq \;\; \mathbf{8}\,n^2$, $\quad n \geq \mathbf{3}$ since $17 \cdot 3 = 51 > 46$, which means

that $-17n + 46 < 0$ for all $n \geq 3$

So we can take $\boldsymbol{c} = \mathbf{8}$ and $\boldsymbol{n_0} = \mathbf{3}$

OBSERVATIONS

Suppose $f(n) = O(g(n))$ then:

- We can find multiple choices of constants to prove it. What matters is that *one* choice exists.

- These constants depend on $f(n)$. A different function belonging to $O(g(n))$ would usually require different constants.

WHAT DOES $O(1)$ MEAN?

We say $f(n)$ is $O(1)$, if there exist two positive constants n_0 and c such that, for all $n \geq n_0$,

$$f(n) \leq c.$$

So it just means that $f(n)$ is bounded by a constant.

BACK TO INSERTION SORT

At the beginning of today's lecture we found the function describing the worst-case running time for insertion sort.

$$T_{worst}(n) = a n^2 + b n + c$$

where a, b, and c are some constants. $a, b > 0$, $c < 0$

Claim: $T_{worst}(n)$ is $O(n^2)$

$T_{worst}(n)$ IS $O(n^2)$ – PROOF

Claim: $T_{worst}(n)$ is $O(n^2)$

Proof: $T_{worst}(n) = an^2 + bn + c$

$\leq an^2 + bn^2$, for all $n \geq \mathbf{1}$ (since $c < 0$)

$= (\boldsymbol{a} + \boldsymbol{b})n^2$

So we can take $\boldsymbol{c' = a + b}$ and $\boldsymbol{n_0 = 1}$.

OBSERVATION ON WORST-CASE UPPER BOUNDS

- When we use asymptotic notation with functions that represent the running time of an algorithm, we need to understand which running time we are referring to. Sometimes we might be interested in the worst-case running time, others in the running time no matter what the input is.

- Since O-notation describes an upper bound, when we use it to bound the worst-case running time of an algorithm, then we have a bound on the running time of the algorithm *on every input*.
  That is,
  Since $T(n) \leq T_{worst}(n)$, if $T_{worst}(n) = O\big(g(n)\big)$, then $T(n) = O(g(n))$

HOW ELSE TO USE THE DEFINITION

We can also use the formal definition to prove that a function $f(n)$ is not $O(g(n))$.

- For example, $6n^3 \notin O(n^2)$.
- *Proof (by contradiction)*: Suppose $6n^3 \in O(n^2)$. Then, by definition there exists two positive constants c and n_0 such that for all $n \geq n_0$

$$6n^3 \leq c \cdot n^2$$

dividing both sides by n^2 and by 6, we get

$$n \leq \frac{c}{6}$$

which cannot possibly be true for arbitrarily large n.

TIGHT BOUNDS

- Since Big O is an upper bound, if $f(n)$ is $O(n)$, then it is also $O(n^2), O(n^3)$, etc.

- That is, $O(n)$ is a subset of $O(n^2)$, which is a subset of $O(n^3)$.

- When we ask for a tight upper bound on $f(n)$ though, we want the simple function $g(n)$ such that $O(g(n))$ is the smallest set that $f(n)$ belongs to.

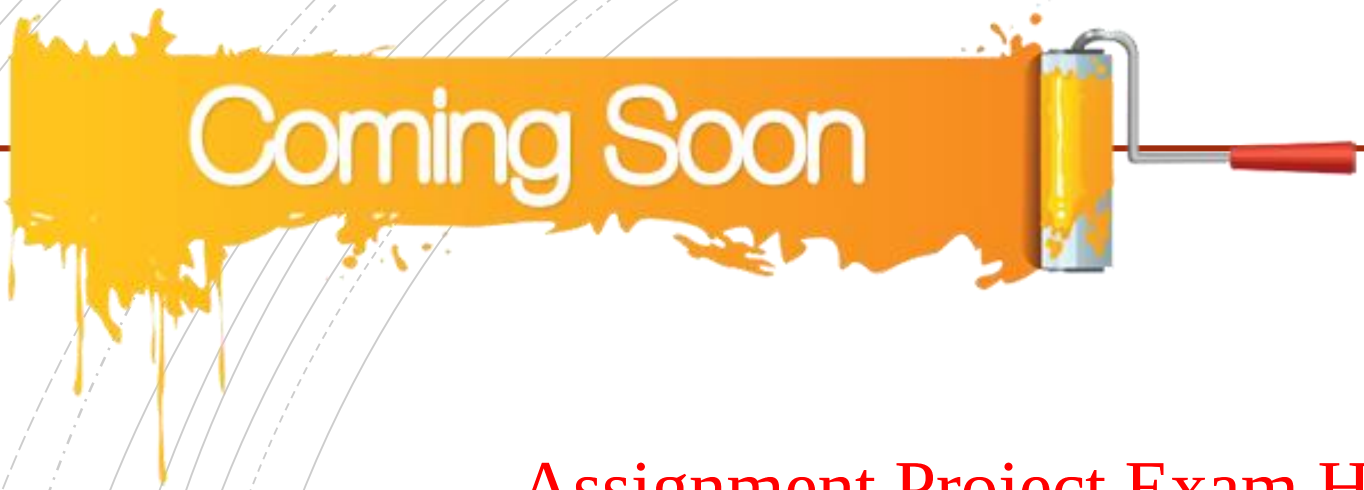
FINAL GENERAL OBSERVATION

Never write $O(3n)$, $O(5\, log_2 n)$, etc.

Instead, write $O(n)$, $O(\log n)$, etc.

Why? The point of O-notation is to *avoid dealing with constant factors*.

It is still *technically* correct to write the above. We just don't do it.



In the next video:

- Big-Omega, $\Omega(\cdot)$

- Big-Theta, $\Theta(\cdot)$