CSUS Help desk is hosting a

JAVA BOOTCAMP

Thursday September 17th from 5:30pm to 7h30pm

This bootcamp is aimed to programmers who don't know the particularities of Java. There will be an overview of syntactic and semantic difference between java and other coding languages, with a focus on OOP (including polymorphism and inheritance).

The zoom link is: https://mcgill.zoom.us/j/92101531362

CSUS

COMP251: Running time analysis and the Big O notation

Jérôme Waldispühl

School of Computer Science

McGill University

Based on slides from M. Langer and M. Blanchette

Outline

- Motivations
- The Big O notation
  - Definition
  - Examples
  - Rules
- Big Omega and Big Theta
- Applications

Measuring the running "time"

- Goal: Analyze an algorithm written in pseudocode and describe its running time
  - Without having to write code
  - In a way that is independent of the computer used
- To achieve that, we need to
  - Make simplifying assumptions about the running time of each basic (primitive) operations
  - Study how the number of primitive operations depends on the size of the problem solved

Primitive Operations

Simple computer operation that can be performed in time that is always the same, independent of the size of the bigger problem solved (we say: constant time)

- **Assigning a value to a variable:** x ←1 T_{assign}
- **Calling a method:** Expos.addWin() T_{call}
  - Note: doesn't include the time to execute the method
- **Returning from a method:** return x; T_{return}
- **Arithmetic operations on primitive types** T_{arith}
  x + y, r*3.1416, x/y, etc.
- **Comparisons on primitive types:** x==y T_{comp}
- **Conditionals:** if (...) then.. else... T_{cond}
- **Indexing into an array:** A[i] T_{index}
- **Following object reference:** Expos.losses T_{ref}

Note: Multiplying two Large Integers is *not* a primitive operation, because the running time depends on the size of the numbers multiplied.

FindMin analysis

```
Algorithm findMin(A, start, stop)
Input: Array A, index start & stop
Output: Index of the smallest element of A[start:stop]
minvalue ← A[start]
minindex ← start
index ← start + 1
while ( index <= stop ) do {
   if (A[index]<minvalue)
   then {
        minvalue ← A[index]
        minindex ← index
   }
   index = index + 1
}
return minindex
```


Running time

$T_{index} + T_{assign}$	
T_{assign}	
$T_{arith} + T_{assign}$	
$T_{comp} + T_{cond}$	repeated
$T_{index} + T_{comp} + T_{cond}$	stop-start
$T_{index} + T_{assign}$	times
T_{assign}	
$T_{assign} + T_{arith}$	
$T_{comp} + T_{cond}$ (last check of loop)	
T_{return}	

Worst case running time

- Running time depends on n = stop – start + 1
  - But it also depends on the content of the array!
- What kind of array of n elements will give the worst running time for findMin?

Example:

5	4	3	2	1	0

- The best running time?

Example:

0	1	2	3	4	5

More assumptions

- Counting each type of primitive operations is tedious
- The running time of each operation is roughly comparable:

$T_{assign} \approx T_{comp} \approx T_{arith} \approx \ldots \approx T_{index} = 1$ primitive operation

- We are only interested in the **number** of primitive operations performed

Worst-case running time for findMin becomes:

$$T(n) = 8 + 10 * n$$

Selection Sort

```
Algorithm SelectionSort(A,n)
Input: an array A of n elements
Output: the array is sorted
i← 0                                          1
while (i<n) do {                              2
    minindex ← findMin(A,i,n-1)               3 + T_FindMin(n-1-i+1)=3 +(10 (n-i) - 2)
    t ← A[minindex]                           2
    A[minindex] ← A[i]                        3
    A[i] ← t                                  2
  i ← i + 1                                   2
}                                             2 (last check of loop condition)
```

Primitive operations (worst case) :

Selection Sort: adding it up

Total: $T(n) = 1 + (\sum_{i=0}^{n-1} 12 + 10 (n - i)) + 2$

$= 3 + (12\ n + 10 \sum_{i=0}^{n-1} (n-i))$

$= 3 + 12\ n + 10 (\sum_{i=0}^{n-1} n) - 10 (\sum_{i=0}^{n-1} i)$

$= 3 + 12\ n + 10\ n * n - 10 ((n-1) * n) / 2)$

$= 3 + 12\ n + 10\ n^2 - 5\ n^2 + 5\ n$

$= 5\ n^2 + 17\ n + 3$

More simplifications

We have: $T(n) = 5\ n^2 + 17\ n + 3$

Simplification #1:

When n is large, $T(n) \approx 5\ n^2$

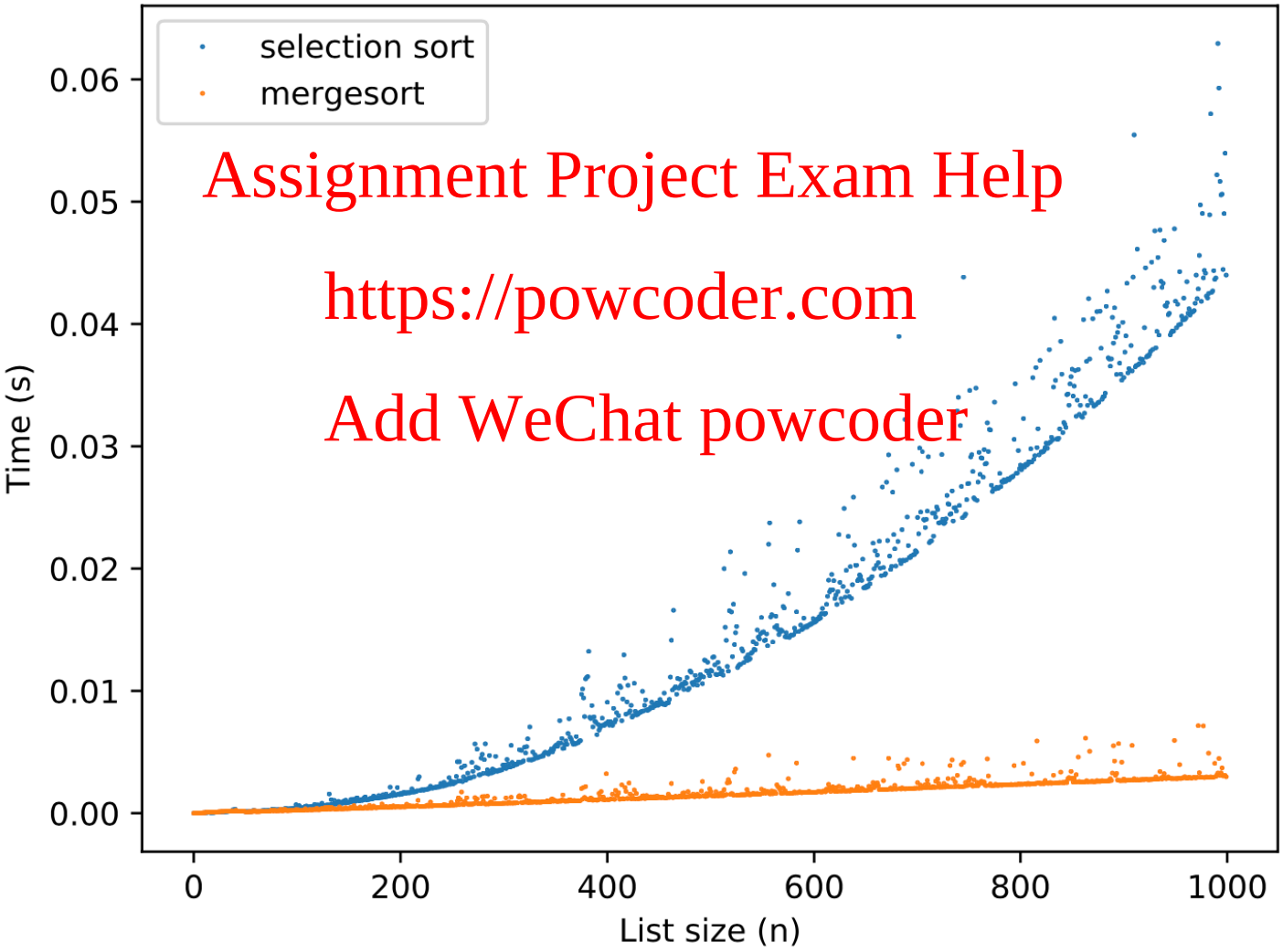
Simplification #2:

When n is large, T(n) grows approximately like n^2

We will write T(n) is $O(n^2)$

"T(n) is big O of n squared"

Asymptotic behavior



Towards a formal definition of big O

Let $t(n)$ be a function that describes the time it takes for some algorithm on input size n.

We would like to express how $t(n)$ grows with n, as n becomes large i.e. *asymptotic* behavior.

Unlike with limits, we want to say that $t(n)$ grows like certain *simpler* functions such as $\sqrt{n}, \log_2 n\,, n, n^2, 2^n$ …

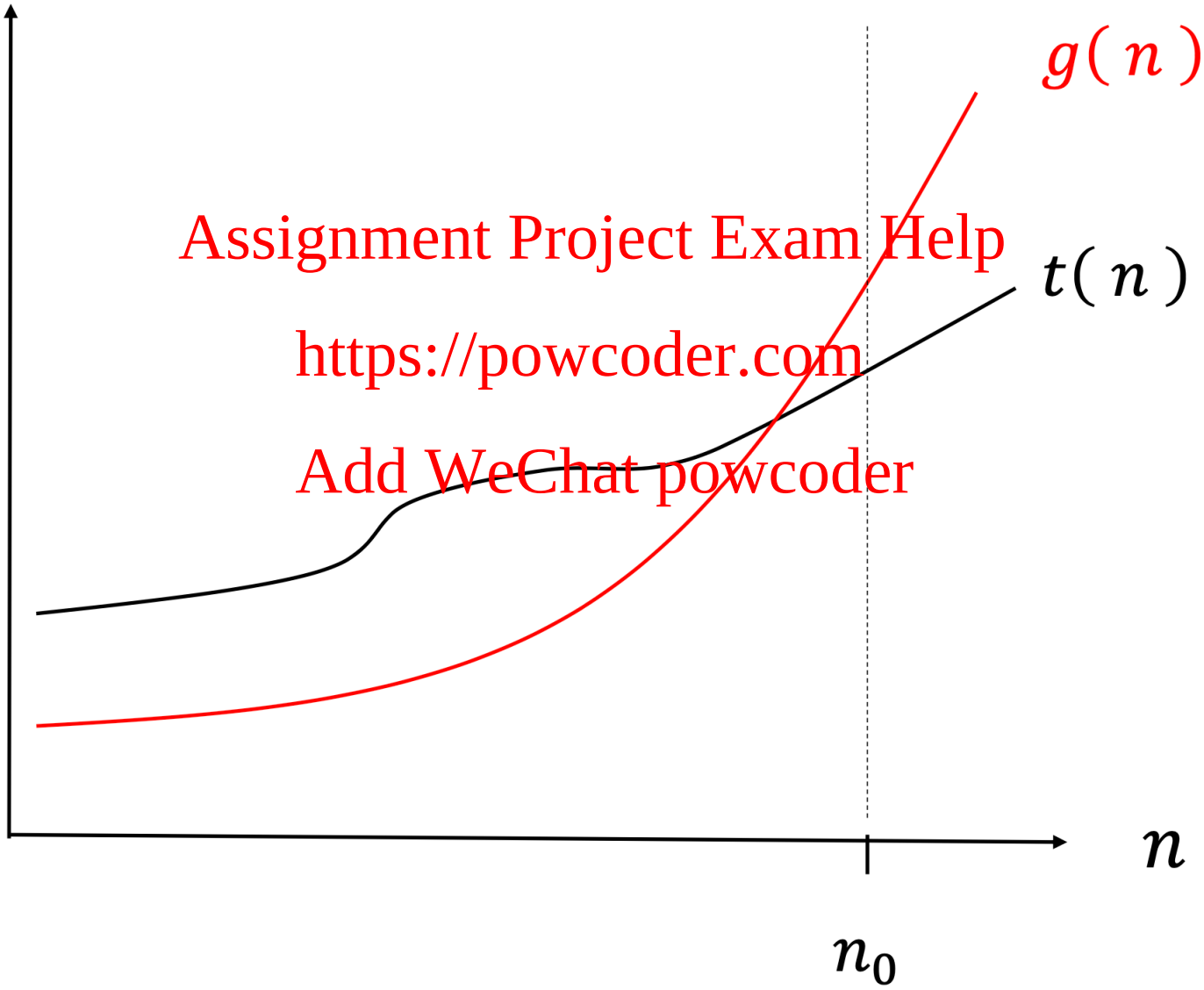
Preliminary Definition

Let $t(n)$ and $g(n)$ be two functions, where $n \geq 0$. We say $t(n)$ is *asymptotically bounded above by* $g(n)$ if there exists n_0 such that, for all $n \geq n_0$,

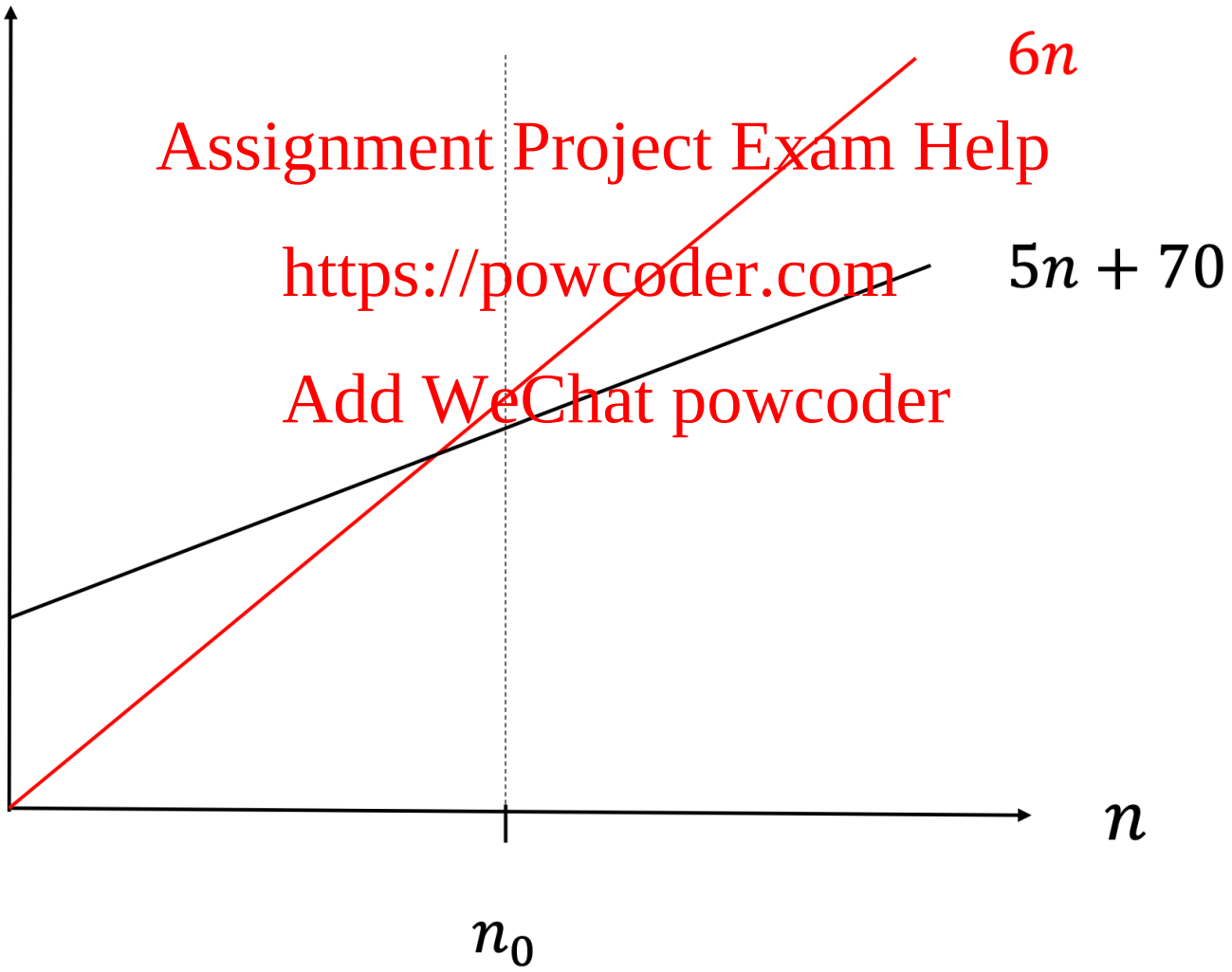
$$t(n) \leq g(n)$$

WARNING: This is not yet a formal definition!

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



Example



Example

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

Proof:

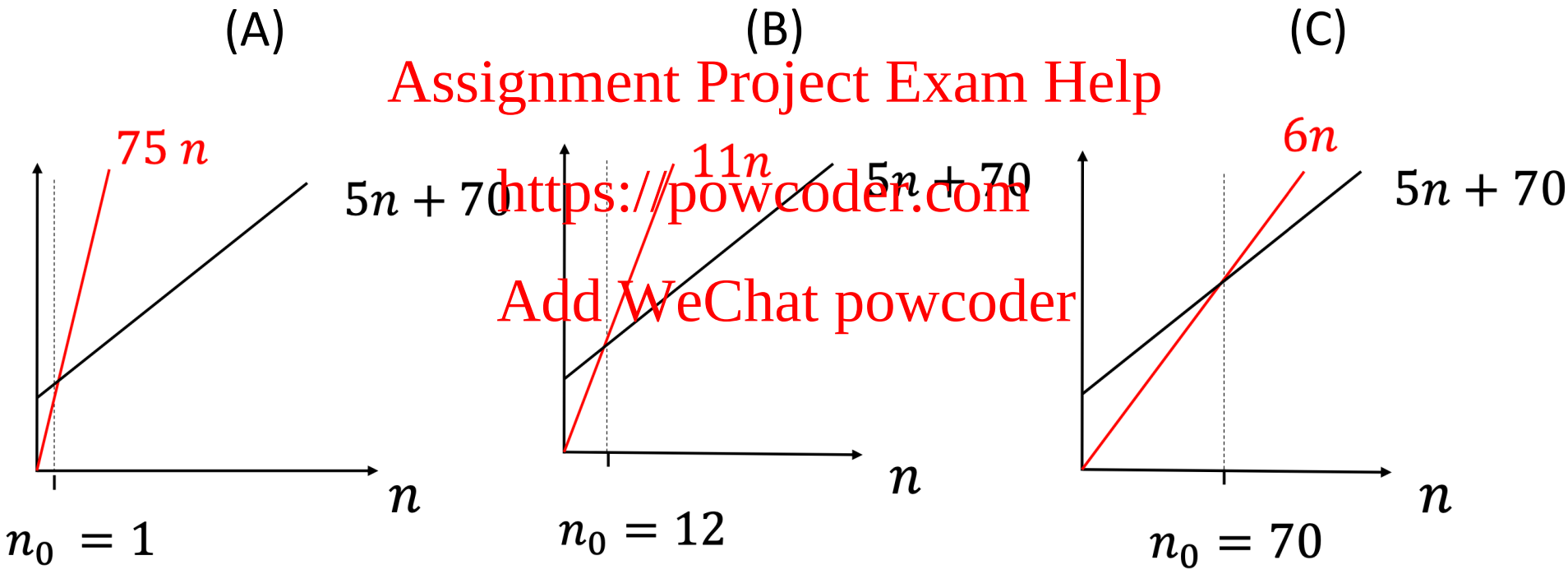
(State definition) We want to show there exists an n_0 such that, for all $n \geq n_0$, $5 \cdot n + 70 \leq 6 \cdot n$.

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

Thus, we can use $n_0 = 70$

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

Choosing a function and constants



Motivation

We would like to express formally how some function $t(n)$ grows with n as n becomes large.

We would like to compare the function $t(n)$ with *simpler* functions , $g(n)$, such as as $\sqrt{n}, \log_2 n\,, n, n^2, 2^n$ …

Formal Definition

Let $t\ n$ and $g\ n$ be two functions, where $n \geq 0$.

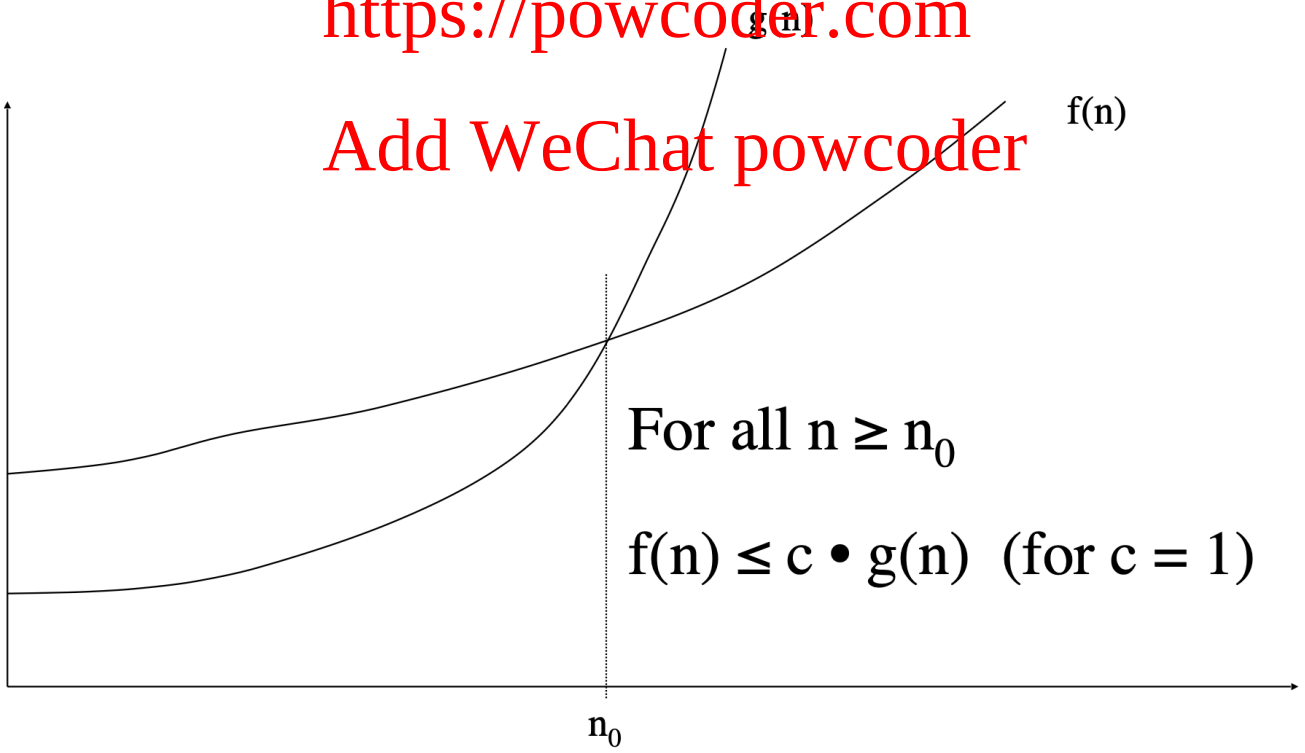
We say $t(n)$ is $O(g(n))$ if there exists two positive constants n_0 and c such that, for all $n \geq n_0$,

$$t(n) \leq c \cdot g(n)$$

Note: $g(n)$ will be a simple function, but this is not required in the definition.

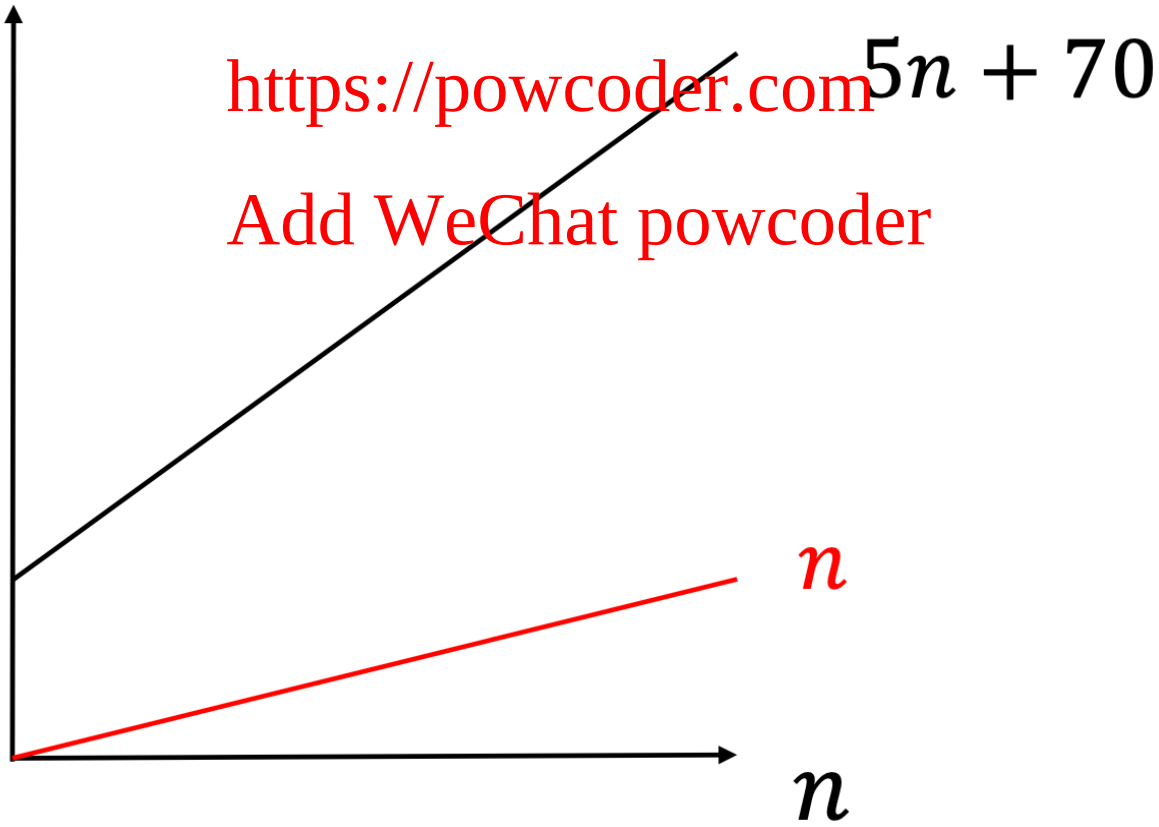
Intuition

"$f(n)$ is $O(g(n))$" if and only if there exists a point n_0 beyond which $f(n)$ is less than some fixed constant times $g(n)$.



Example (1)

Claim: $5 \cdot n + 70$ *is* $O(n)$



Proof(s)

Claim: $5 \cdot n + 70 \; is \; O(n)$

Proof 1: $5 \cdot n + 70 \leq 5 \cdot n + 70 \cdot n = 75 \cdot n, \; if \; n \geq 1$

Thus, take $c = 75$ and $n_0 = 1$.

Proof 2: $5 \cdot n + 70 \leq 5 \cdot n + 6 \cdot n = 11 \cdot n, \; if \; n \geq 12$

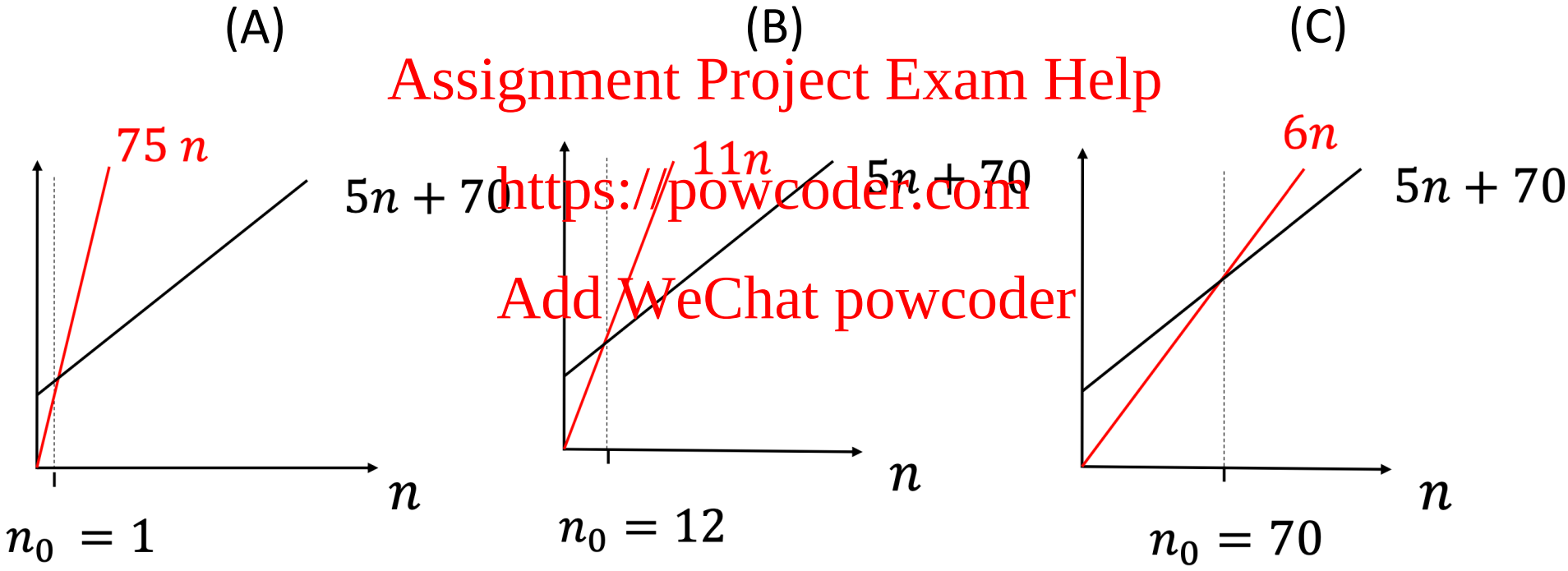
Thus, take $c = 11$ and $n_0 = 12$.

Proof 3: $5 \cdot n + 70 \leq 5 \cdot n + n = 6 \cdot n, \; if \; n \geq 70$

Thus, take $c = 6$ and $n_0 = 70$.

All these proofs are correct and show that $5 \cdot n + 70$ is $O(n)$

Visualization



Example (2)

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

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

$\leq 54n^2$

Thus, we can take $c = 54$ and $n_0 = 1$.

Proof 2: $8n^2 - 17n + 46 \leq 8n^2$, if $n \geq 3$

Thus, we can take $c = 8$ and $n_0 = 3$.

What does O(1) mean?

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

$$t(n) \leq c$$

So, it just means that $t(n)$ is bounded.

Tips

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

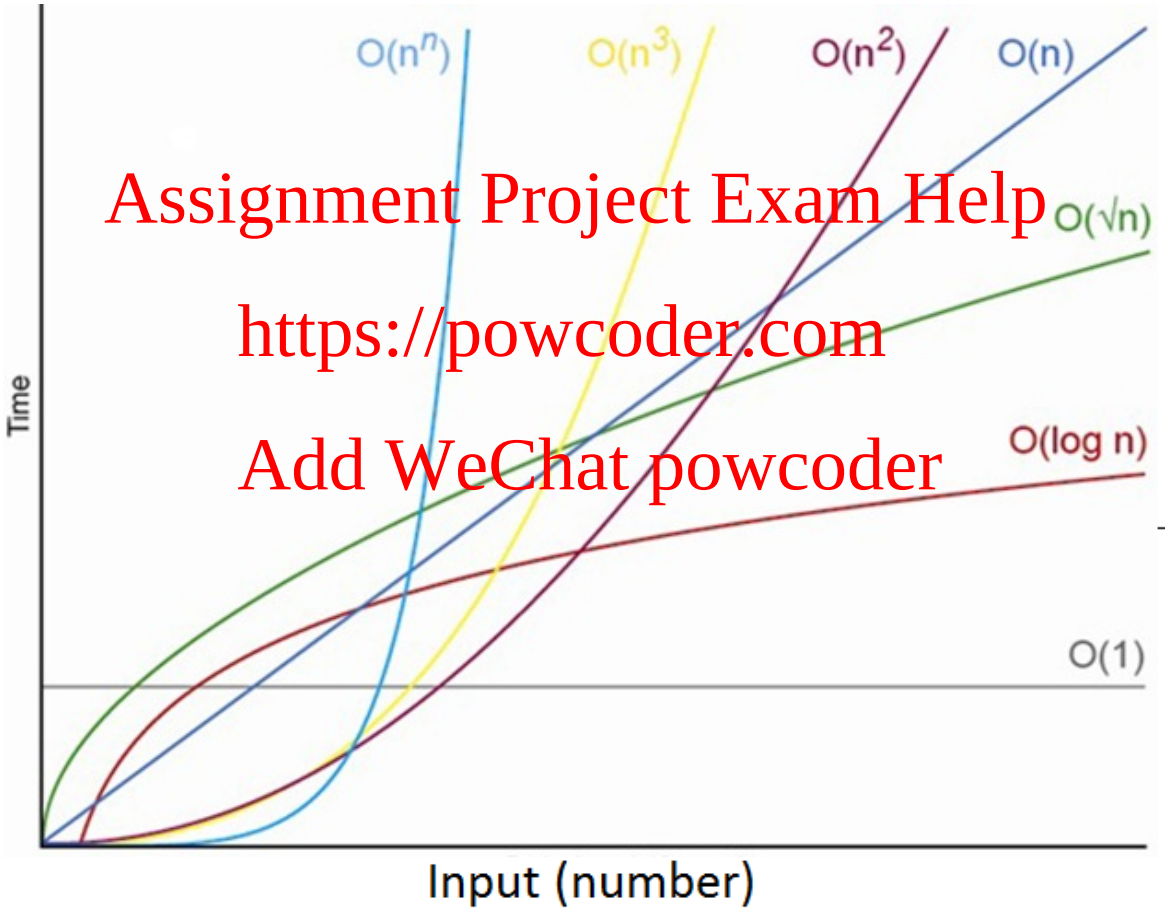
Instead, write $O(n), O(\log_2 n), etc.$

Why? The point of the big O notation is to avoid dealing with constant factors. It's technically correct but we don't do it…

Other considerations

- n_0 and c are not *uniquely* defined. For a given n_0 and c that satisfies $O()$, we can increase one or both to again satisfy the definition. **There is not "better" choice of constants.**
- **However,** we generally want a "tight" upper bound (asymptotically), so functions in the big O gives us more information (Note: This is not the same as smaller n_0 or c). For instance, $f(n)$ that is $O(n)$ is also $O(n^2)$ and $O(2^n)$. But $O(n)$ is more informative.

Growth of functions



(from stackoverflow)

Tip: It is helpful to memorize the relationship between basic functions.

Practical meaning of big O…

	constant	*logarithmic*	*linear*	*N-log-N*	*quadratic*	*cubic*	*exponential*
n	O(1)	O(log *n*)	O(*n*)	O(*n* log *n*)	O(n^2)	O(n^3)	O(2^n)
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	1.84×10^{19}

If the unit is in seconds, this would make ~10^{11} years…

Constant Factor rule

Suppose f(n) is O(g(n)) and a is a positive constant.
Then, $a \cdot f(n)$ is also $O(g(n))$.

Proof: By definition, if f(n) is O(g(n)) then there exists two positive constants n_0 and c such that for all $n \geq n_0$,

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

Thus, $$a \cdot f(n) \leq a \cdot c \cdot g(n)$$

We use the constant $a \cdot c$ to show that $a \cdot f(n)$ is $O(g(n))$.

Sum rule

Suppose $f_1(n)$ *is* $O(g(n))$ and $f_2(n)$ *is* $O(g(n))$.

Then, $f_1(n) + f_2(n)$ *is* $O(g(n))$.

Proof: Let n_1, c_1 and n_2, c_2 be constants such that

$$f_1(n) \leq c_1 g(n), \text{ for all } n \geq n_1$$

$$f_2(n) \leq c_2 g(n), \text{ for all } n \geq n_2$$

So, $f_1(n) + f_2(n) \leq (c_1 + c_2) g(n), \text{ for all } n \geq \max(n_1, n_2)$.

We can use the constants $c_1 + c_2$ and $\max(n_1, n_2)$ to satisfy the definition.

Generalized Sum rule

Suppose $f_1(n)\ is\ O(g(n))$ and $f_2(n)\ is\ O(g(n))$.

Then, $f_1(n) + f_2(n) is\ O(g_1(n) + g_2(n))$.

Proof: Exercise…

Product Rule

Suppose $f_1(n)$ *is* $O(g(n))$ and $f_2(n)$ *is* $O(g(n))$.

Then, $f_1(n) \cdot f_2(n)$ *is* $O(g_1(n) \cdot g_2(n))$.

Proof: Let n_1, c_1 and n_2, c_2 be constants such that

$$f_1(n) \leq c_1 g_1(n), \text{ for all } n \geq n_1$$

$$f_2(n) \leq c_2 g_2(n), \text{ for all } n \geq n_2$$

So, $f_1(n) \cdot f_2(n) \leq (c_1 \cdot c_2) \cdot (g_1(n) \cdot g_2(n))$, for all $n \geq \max(n_1, n_2)$.

We can use the constants $c_1 \cdot c_2$ and $\max(n_1, n_2)$ to satisfy the definition.

Transitivity Rule

Suppose $f(n)$ *is* $O(g(n))$ and $g(n)$ *is* $O(h(n))$.

Then, $f(n)$ *is* $O(h(n))$.

Proof: Let n_1, c_1 and n_2, c_2 be constants such that

$$f(n) \leq c_1 g(n), \text{ for all } n \geq n_1$$

$$g(n) \leq c_2 h(n), \text{ for all } n \geq n_2$$

So, $f(n) \leq (c_1 \cdot c_2) h(n), \text{ for all } n \geq \max(n_1, n_2)$.

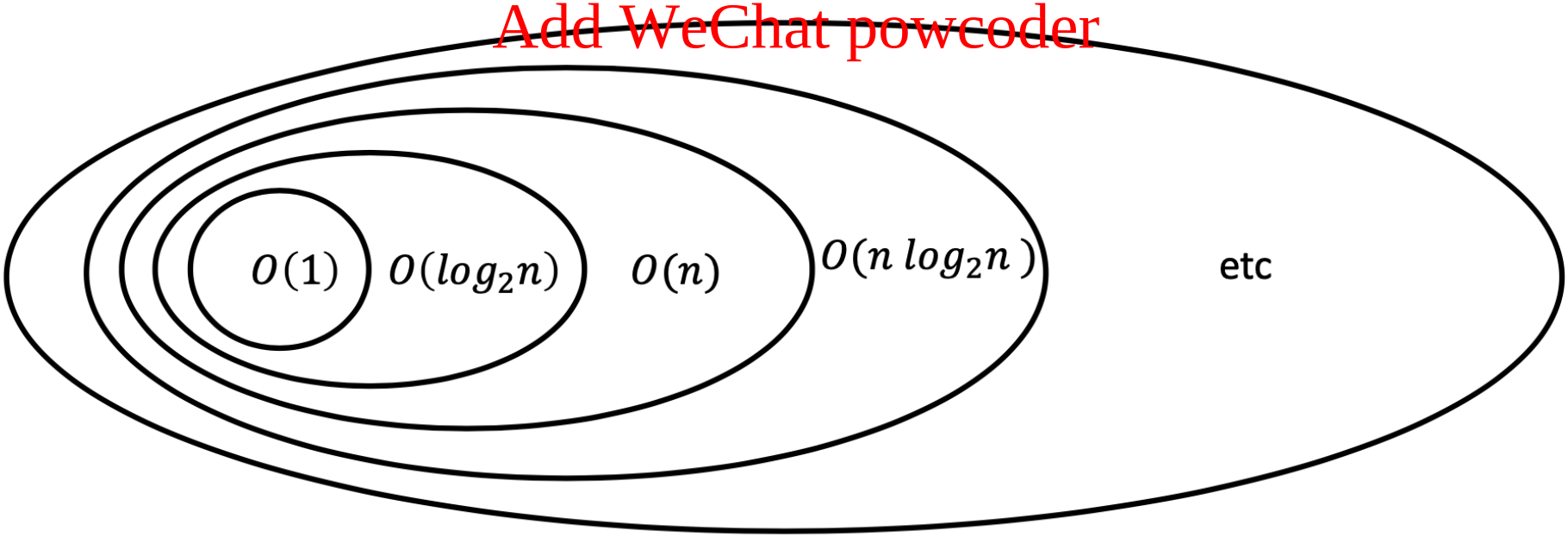
We can use the constants $c_1 \cdot c_2$ and $\max(n_1, n_2)$ to satisfy the definition.

Notations

If $f(n)$ is $O\big(g(n)\big)$, we often write $f(n) \in O(g(n))$. That is a member of the functions that are $O\big(g(n)\big)$.

For n sufficiently large we have, $1 < \log_2 n < n < n \log_2 n$ …

And we write $O(1) \subset O(\log_2 n) \subset O(n) \subset O(n \log_2 n)$ …



The Big Omega notation (Ω)

Let $t(n)$ and $g(n)$ be two functions with $n \geq 0$.

We say $t(n)$ is $\Omega(g(n))$, if there exists two positives constants n_0 and c such that, for all $n \geq n_0$,

$$t(n) \geq c \cdot g(n)$$

Note: This is the opposite of the big O notation. The function g is now used as a "*lower* bound".

Example

Claim: $\frac{n(n-1)}{2}$ is $\Omega(n^2)$.

Proof: We show first that $\frac{n(n-1)}{2} \geq \frac{n^2}{4}$.

$$\Leftrightarrow 2n(n-1) \geq n^2$$

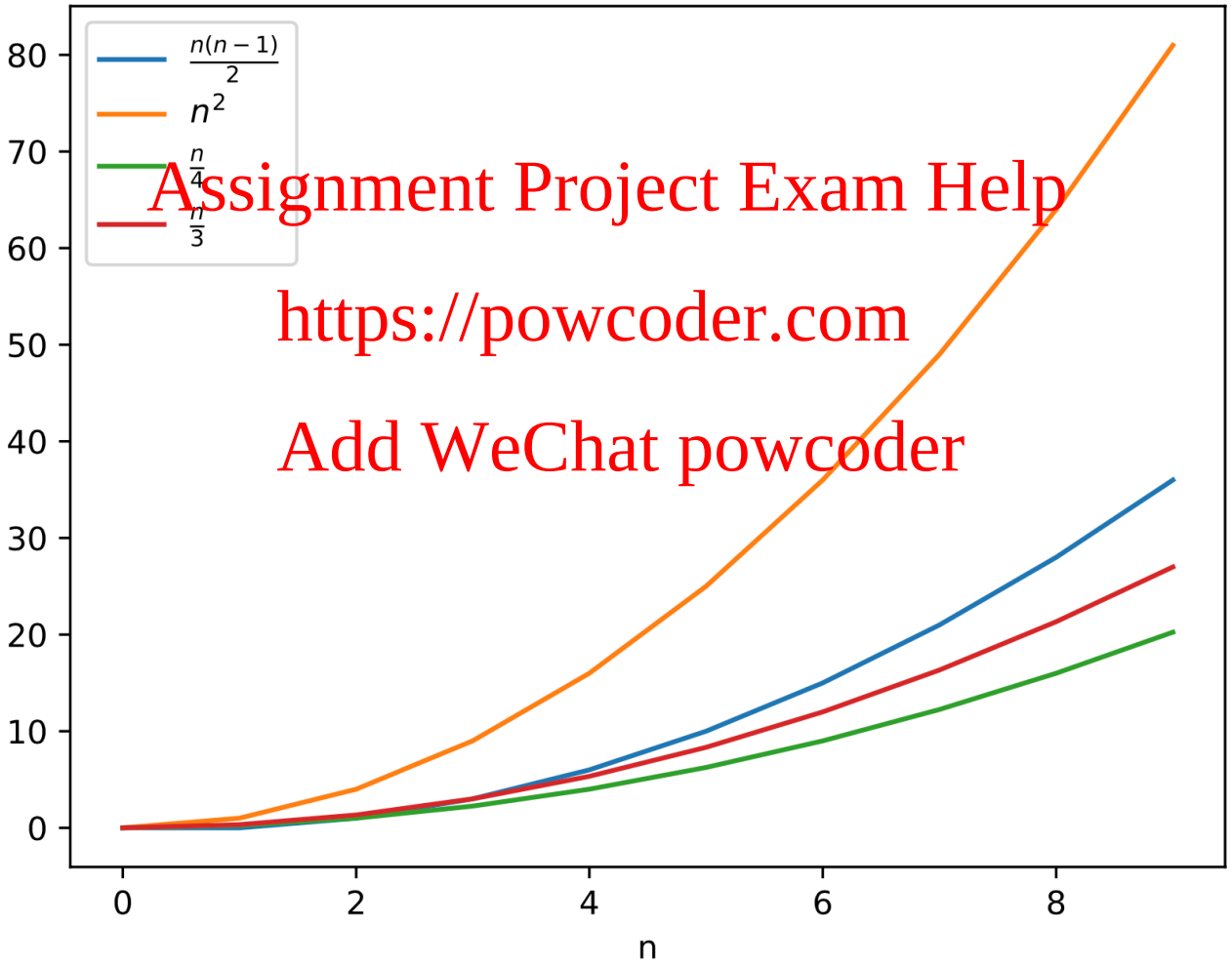
$$\Leftrightarrow n^2 \geq 2n$$

$$\Leftrightarrow n \geq 2$$

Thus, we take $c = \frac{1}{4}$ and $n_0 = 2$.

(Exercise: Prove that it also works with $c = \frac{1}{3}$ and $n_0 = 3$.

Intuition

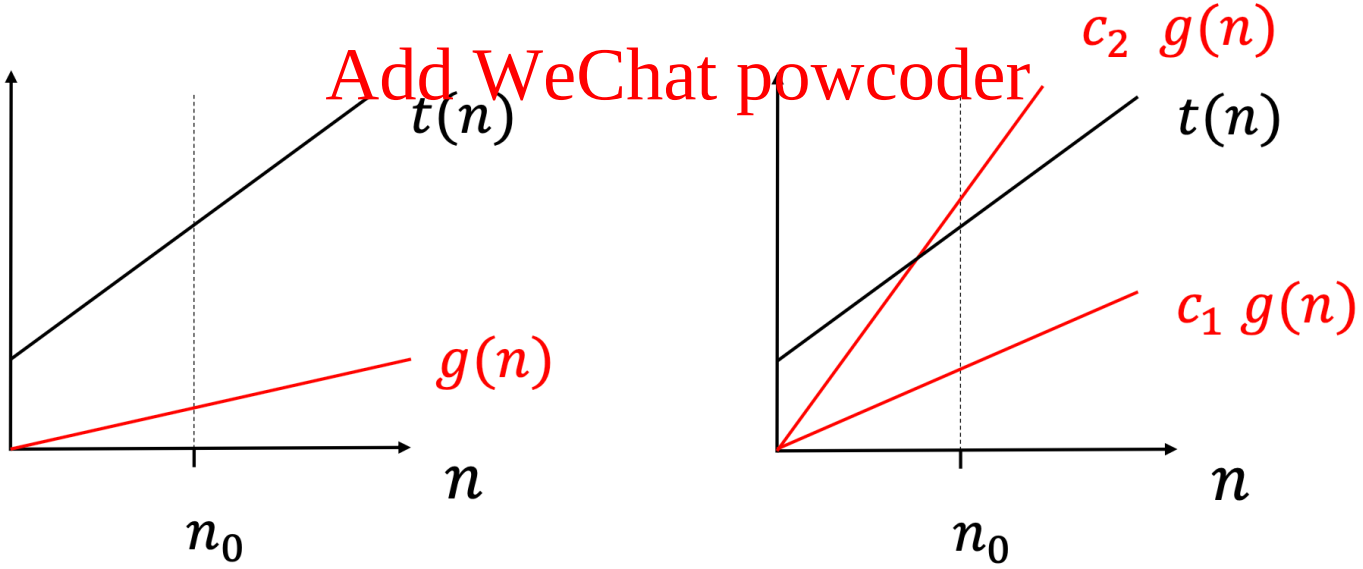


And… big Theta!

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

We say $t(n)$ is $\Theta(g(n))$ if there exists three positive constants n_0 and c_1, c_2 such that, for all $n \geq n_0$,

$$c_1 \cdot g(n) \leq t(n) \leq c_2 \cdot g(n)$$



Note: if $t(n)$ is $\Theta(g(n))$. Then, it is also $O(g(n))$ and $\Omega(g(n))$.

Example

Let $t(n) = 4 + 17\log_2 n + 3n + 9n\log_2 n + \frac{n(n-1)}{2}$

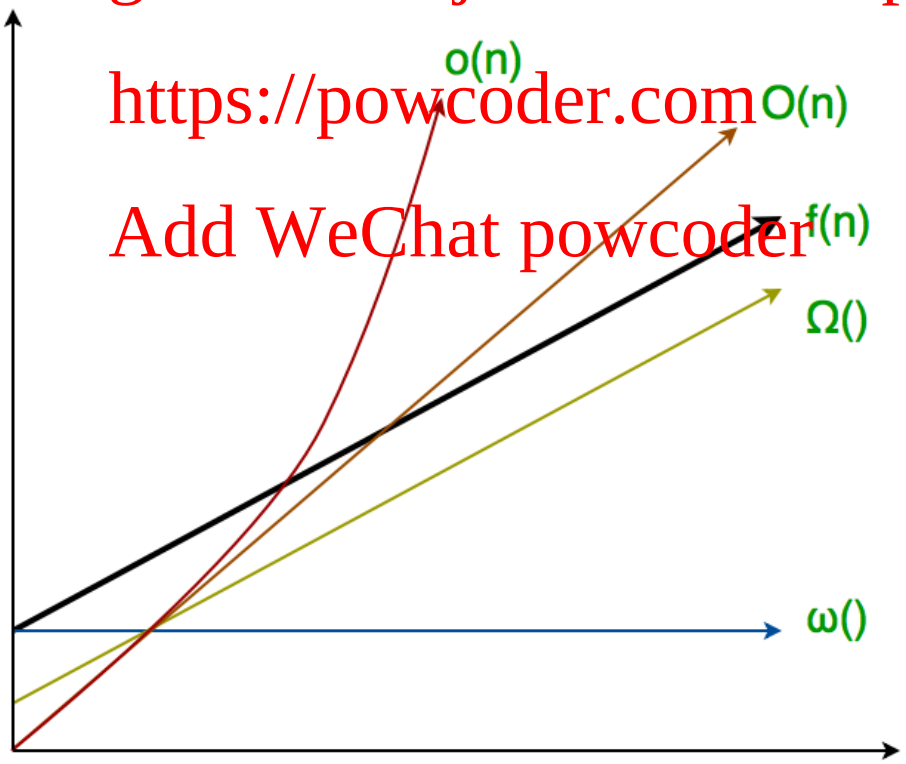
Claim: $t(n)$ is $\Theta(n^2)$

Proof:

$$\frac{n^2}{4} \leq t(n) \leq (4 + 17 + 3 + 9 + \frac{1}{2}) \cdot n^2$$

Big vs. little

The big O (resp. big Ω) denotes a tight upper (resp. lower) bounds, while the little o (resp. little ω) denotes a lose upper (resp. lower) bounds.



(from geekforgeek.org)

Back to running time analysis

The time it takes for an algorithm to run depends on:

- constant factors (often implementation dependent)
- the size n of the input
- the values of the input, including arguments if applicable…

Q: What are the best and worst cases?

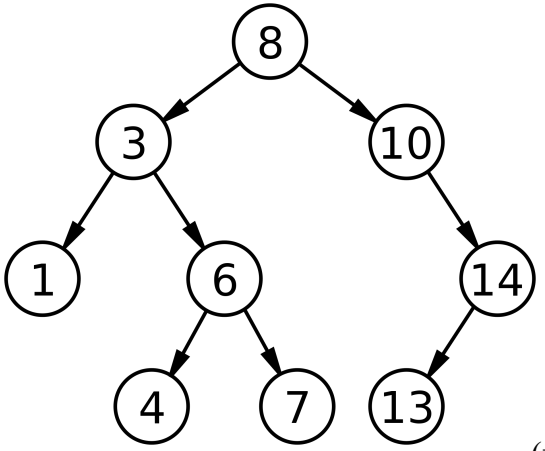
Example (Binary Search)

Best case: The value is exactly in the middle of the array.

$$\Rightarrow \Omega(1)$$

Worst case: You recursively search until you reach an array of size 1 (Note: It does not matter if you find the key or not).

$$\Rightarrow O(\log_2 n)$$



(from Wikipedia)