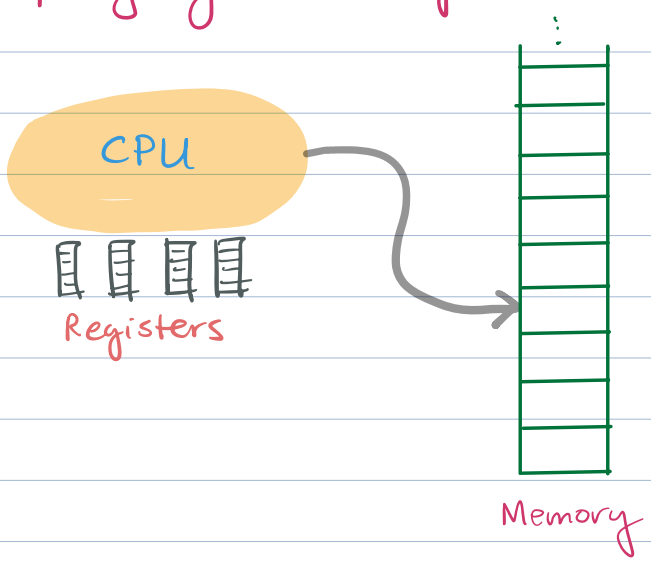
# Complexity Analysis

When it comes to analyzing the efficiency of an algorithm, we are primarily concerned with **Time**

... but clock-ticks vary from processor to processor and we want our analysis of the **algorithm** to be independent of the chip it runs on.

## Simplifying Assumptions




## Random Access Machine (RAM)

- Assumes that each memory location can be accessed in one constant-time step

O(1)

constant time

Further assumptions:

- each read-write of one word is $O(1)$
- each + − * / is $O(1)$
- each == < > || && is $O(1)$

When determining run-time, we "count" all these operations as if they each take same amount of time, but they can be quite different.

e.g. / takes 1000x as many basic (assembly) operations as ||

... but we nevertheless put all these operations into same category of taking "constant time" (as long as the operands fit into a single register, or a constant number of registers...)

A consequence of this way of thinking:

- ten $O(1)$ operations together are also $O(1)$

$$O(1) + O(1) + O(1) + O(1) = O(1)$$

Should we use = here?

In my CSC1 260, we say

$$O(1) + O(1) + O(1) + O(1) \in O(1)$$

$$4 \cdot O(1) \in O(1)$$

if a function is in $O(1)$, then so is any constant multiple of that function

$O(1)$ is a class of functions

**What kind of functions are we trying to classify?**

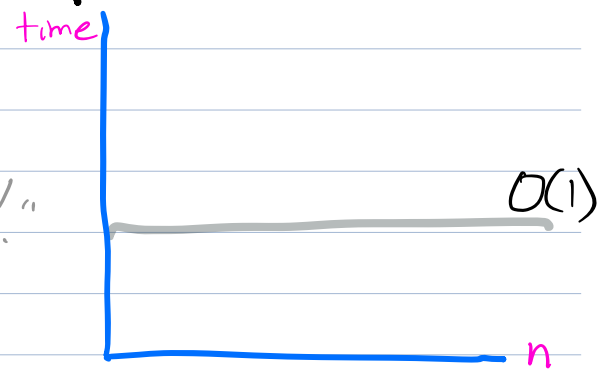
The Running Time of an algorithm, as a function of input size

n bits (or words), $n \geq 0$

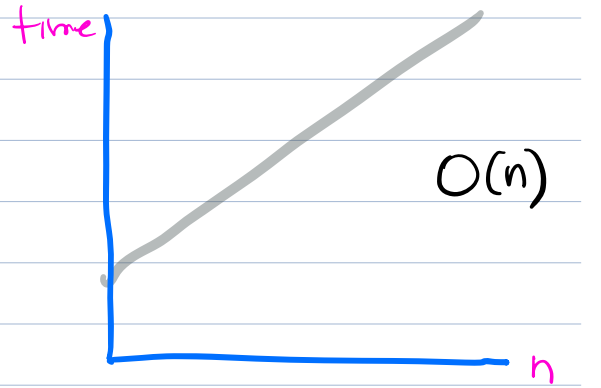
... and is positive-valued $\forall n$ in the range

# What are we trying to capture?

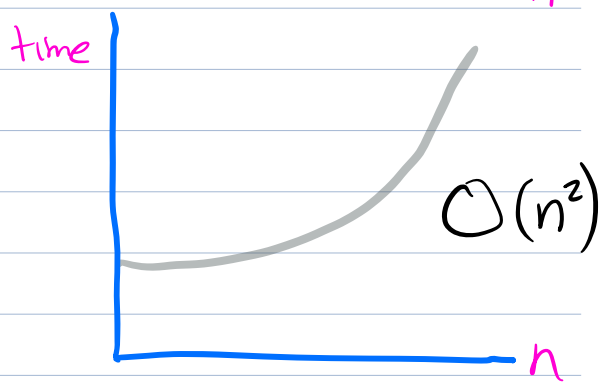
"When my input size (n) doubles / triples / gets huge, the running time stays same."



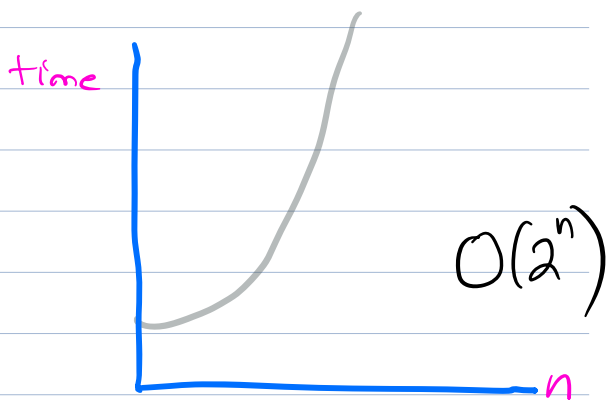

"When my input doubles, my running time doubles."



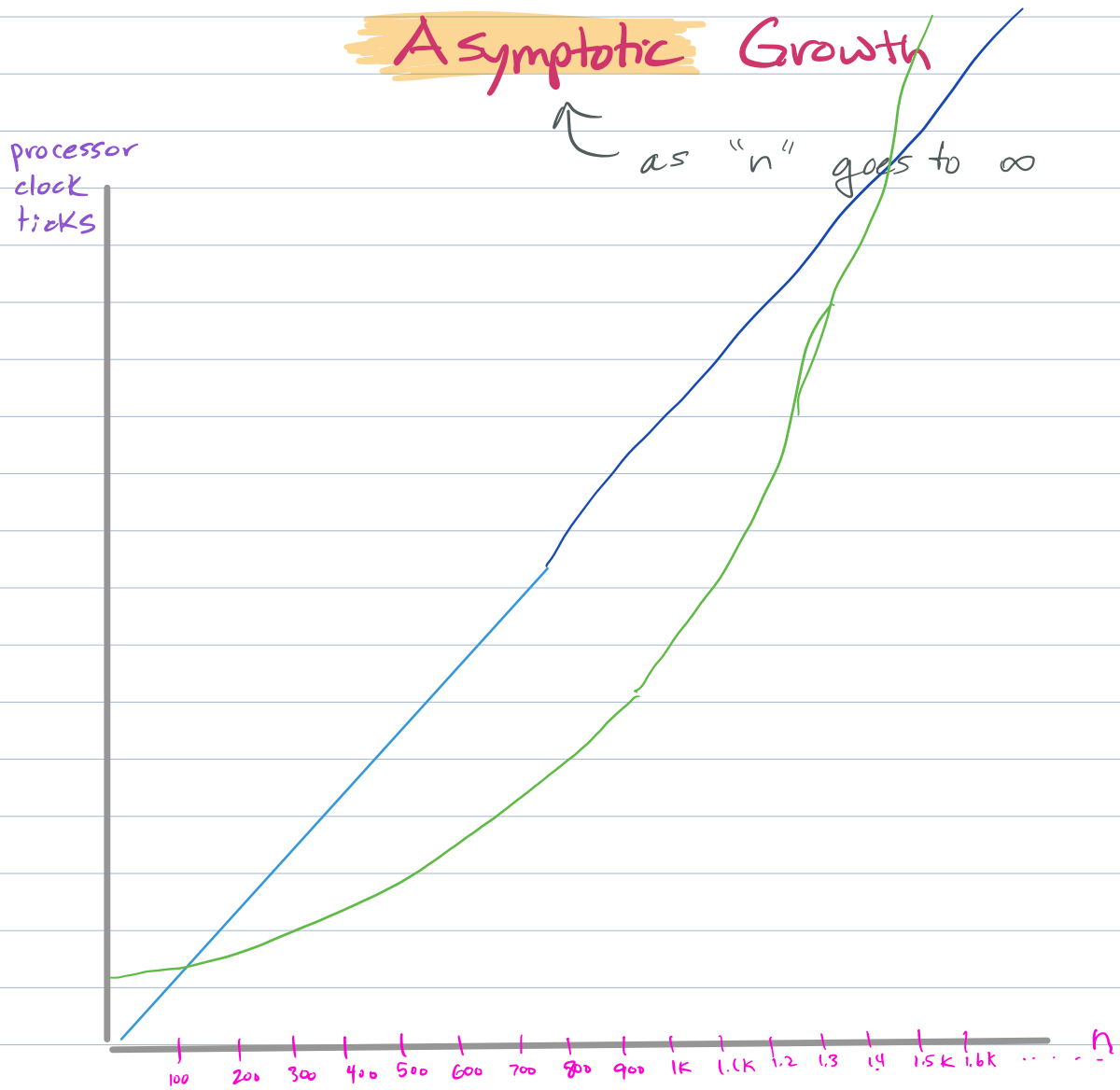

"When my input size doubles, my running time quadruples."



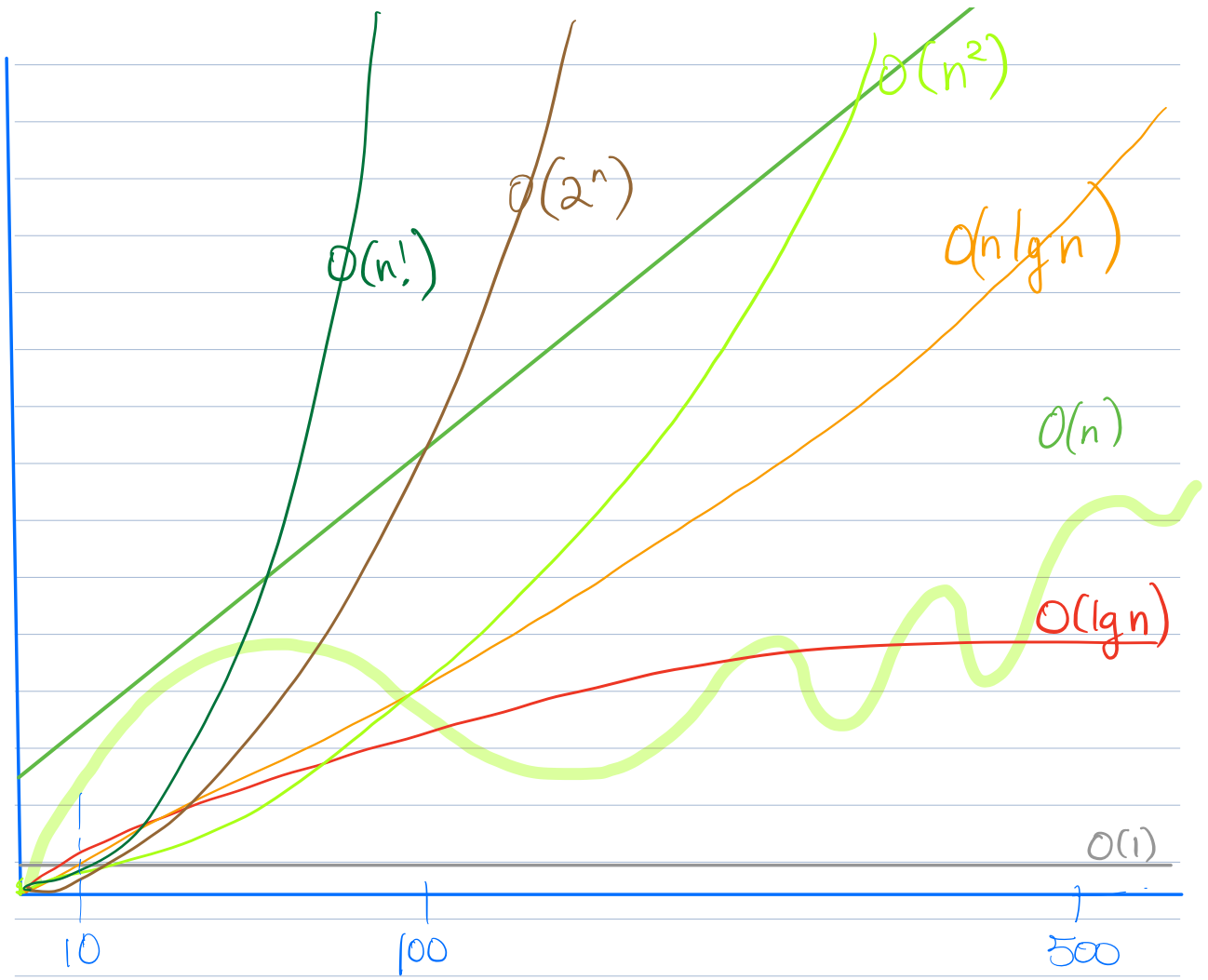

"When my input size increases by 1, my running time doubles."



# Big-O: Classifying functions by their Asymptotic Growth




We want to categorize functions (of n, the input size) so that $f(n) \in O(g(n))$ when $g(n)$ operates like an upper bound on $f(n)$ when $n$ gets LARGE.




Which functions grow asymptotically slower than others?

Order them by "faster run-time" ≈ "slower growth"

Does the order change if you
- add a constant?
- multiply by a constant factor?

# Big-O

Captures: constant factors don't matter
addition of lower terms don't matter.

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \qquad f(n) \in O(g(n))$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} =$ a constant $\qquad f(n) \in O(g(n))$ and $g(n) \in O(f(n))$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \qquad g(n) \in O(f(n))$

That is true but that is not the definition of Big O.

Def$^n$: $f(n) \in O(g(n))$ if $\exists$ constants $c > 0$ and $n_0 \geq 0$

Such that $f(n) \leq c \cdot g(n) \quad \forall n \geq n_0$

What happens when you add two functions?

# How do we use Big O to describe running time of an algorithm?

We say a running time is $\in O(n^2)$

if it is always upper bounded by some multiple of $n^2$, when $n$ is large enough.

$3n^2 + 15 - 100 \quad \in O(n^2)$

$1000\ n^2 \quad \in O(n^2)$

$\frac{1}{3}\ n - 5 \quad \in O(n^2)$

Note: since we are dealing with running times, our functions are assumed to be positive-valued for all $n$

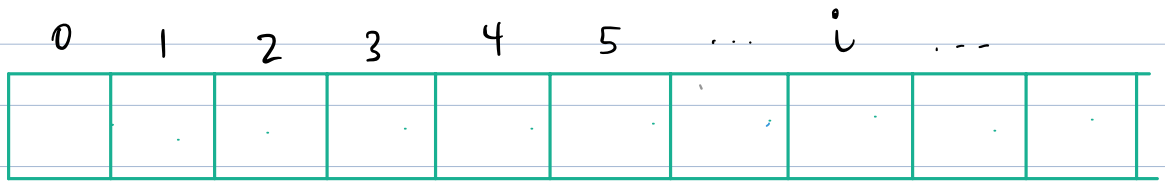
# ADT Array

Operations are:

init() - initializes a new array. Old one is destroyed. New one has no elements in it.

set(int i, float val) - next "get(i)" will return val

float get(int i)
- if there has been no "set(i, val)" since last "init()", returns sentinal value UNDEF
- else returns val of most recent "set(i, val)"



## Algorithms + DS

- can compute an output for a given input

- can be a DS with a lifetime
  operations can change it, or run
  algorithms on it.

---

## Array ADT

- obvious implementation (init is not O(1))
  - use a sentinel.

  - don't use a sentinel.