Clearly Alg 2 is more complicated than Alg 1.

Let's consider the same example instances we did for Alg 1 but for Alg 2.

1. $L=[3]$, $x=3$.
2. $L=[3,4,\ldots,2^{100}]$, $x=3$
3. $L=[1,\ldots,10]$, $x=10^{100}+1$
4. $L=[1,2,\ldots,2^{100}]$, $x=2^{100}+1$
5. $L=[3,4,\ldots,10^{100}]$, $x=1$.

$2^{100} \approx 10^{30}$

million trillion trillion.

(New) 6. $L=[1,\ldots,2^{100}]$, $x=?$ (something inside).

Which algorithm is better?

Although Alg 2 is not always faster than Alg 1, Alg 2 seems to perform better than Alg 1 except in seemingly special situations.

How do we make this more precise?

<u>Computational complexity theory</u> studies the resources used by algorithms solving computational problems.

The most common resource is "computational time".

Another example is memory / space.

How can one even measure computational time?

- On different inputs, we expect very different times.
- On different machines, we expect very different times.

We solve the first problem by ~~taking~~ using a specific analysis. We will use worst-case complexity. The computational ~~comp~~ complexity of an algorithm depends on its worst-case inputs. (So one must think about the kinds of ~~#~~ inputs where the algorithm will perform its worst.)

We solve the second problem by ① measuring "primitive operations" — these are operations that we assume take a constant amount of time — and ② analyzing asymptotic behavior.

~~Asymptotic behavior~~

Let's consider wort-case inputs of Simple Search and Binary Search. Assume we have a list L of length n.

Simple Search (L) : looks at all n elements.

Binary Search (L) : looks at $\log_2 n$ elements.

Assuming that looking up (or getting) an element of a sorted list is a primitive operation, we would say that Simple Search ~~uses~~ uses $O(n)$ operations and Binary Search uses $O(\log n)$.

(This is prounonced "Oh of n" and "Oh of log n".)

Note if we restrict to sorted-lists of length $\leq 8 = 2^3$ then

Simple Search uses ≤ 8 operations
Binary Search uses ≤ 3 operations

Technically a constant factor (independent of input) times both of those.

This covers things like computing "mid" in BinarySearch and ~~constructing~~ getting the length of a sorted-list.

If we increase the size of the ~~leng~~ lists to 2^{100}

Simple Search uses $\leq 2^{100}$ operations,
Binary Search uses ≤ 100 operations.

Complexity Theory:

Complexity theory gives us a means to analyze aspects of algorithms so that we may compare.

We will use worst-case analysis, but there are other popular ones: average-case and amortized.

We will measure only "runtime" by considering primitive operations.

By asymptotics, we mean to record the number of operations as we increase the size of the input. Typically it is useful to think of input as variable-length, say, n. But n is growing larger and larger.

Big O notation.

Big O notation is a shorthand to describe a precise asymptotic statement.

Def. Suppose $f: \mathbb{R} \to \mathbb{R}$ and $g: \mathbb{R} \to \mathbb{R}$ are functions with nonnegative output. We write

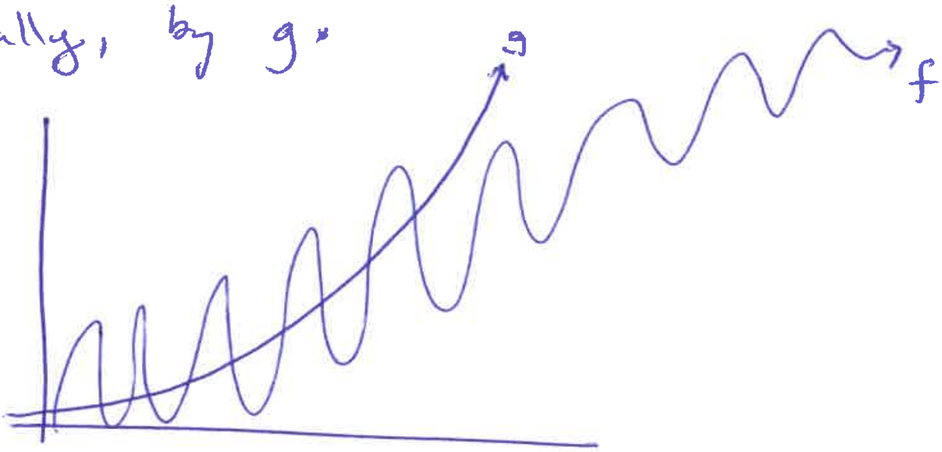
$$f = O(g)$$

if there exists a constant $C \geq 0$ and an x_0 (real number) such that for all $x \geq x_0$

$$f(x) \leq C \cdot g(x).$$

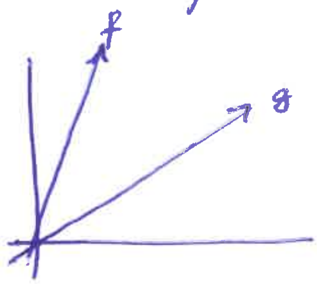
(simplified)
Plain English: The function f is bounded above, asymptotically, by g.

Example of $f = O(g)$



Be careful $f=O(g)$ is an asymptotic statement!

If $f=5x$ and $g=x$, then $f=O(g)$.



And $g=O(f)$.

Plain English: The growth rate of f is not larger than the growth rate of g.

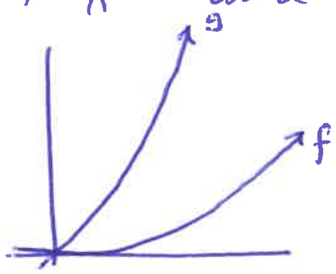
Ex: What about $f=x$ and $g=x^2$?

For $C=1$ and $x_0=1$, we have ~~that~~

$$x \leq x^2$$

for all $x \geq x_0$. So $x=O(x^2)$

Ex: $f=x^2$ and $g=2x^2+x$.



- Clearly $x^2 \leq x^2+x$ for $x \geq 1$. So $x^2=O(x^2+x)$
- Note that $x^2+x \leq \underset{C}{2}x^2$ for $x \geq 1$. So $x^2+x=O(x^2)$.

<u>Fact</u>. If $f = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$ is a polynomial, then $f = O(x^n)$.

So when thinking asymptotically, we can "replace" ~~the~~ a general polynomial with its leading term (the term with the highest exponent).

<u>Fact</u>. Suppose n and m are ~~positive~~ nonnegative integers. Then $x^m = O(x^n)$ if and only if $m \leq n$.

Coming back to primitive operations: they are operations that run in $O(1)$ time — "constant time." In other words, if we increase the input size, the particular operation still takes the same amount of time.

<u>Note</u>: $10^{10^{10^{10}}} = O(1)$, so "constant-time" is not necessarily "fast".

We have a hierarchy of growth rates:

$$O(1) < O(x) < O(x^2) < O(x^3) < \cdots$$

What else is there? (Are all mathematical functions given by a polynomial?!)

<u>Exponential</u>: Recall $\log_b b^x = x$, so