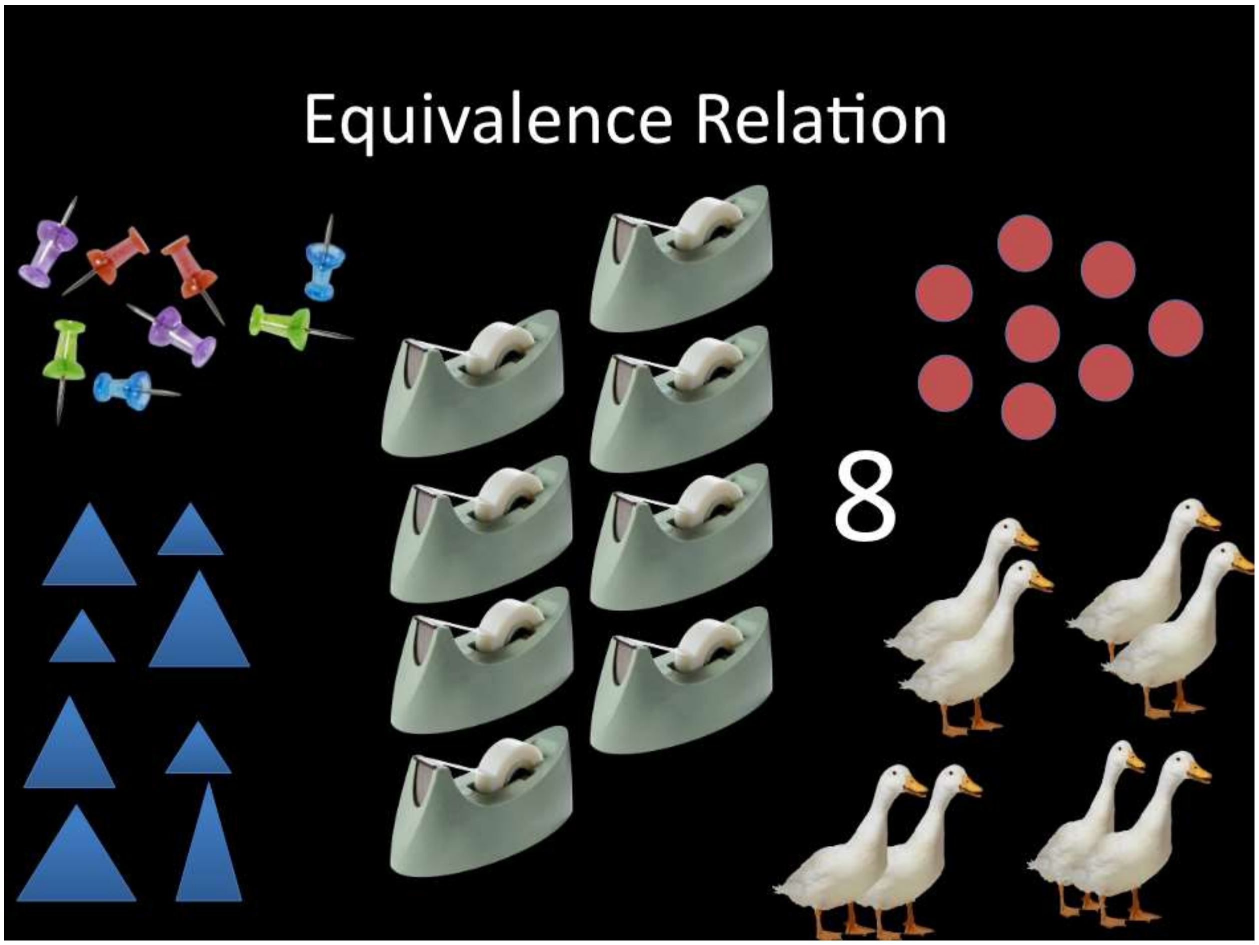
Equivalence Relations Topics:

Relations in computer science are important as they allow us to define relationships in data - students/classes/books/ages/shoe sizes/cousins etc. But a fundamental and essential relation is the equivalence relationship with a set. While all elements of a set may be distinct, we may define a relationship the treats some elements as the same. For example, all integers are distinct and unique, but under the modulo relationship 4, 8, 16, are fundamentally the same thing. This allows us to partition ALL integers under mod 4 into 4 elements. 0, 1, 2, and 3.

The very idea of number is an important equivalence. Counting numbers can be formally defined as the cardinality of a set (remember set theory)?

While the groups of objects in the picture below are all different, there is a sameness about the groups. That sameness is “eight.”



This means that we can understand if 4 geese + 4 geese = 8 geese, then 4 pins + 4 pins = 8 pins and in general 4 + 4 = 8. This may seem trivial, but it is a foundational concept in mathematics.

Consider the sets of all things. Define a relation on this set as the cardinality (number of things) in the set.

So is it reflexive? Number of things in set = number of things in set- yes.
Symmetric ? if number of geese = number of red dots, then number of red dots = Number of geese
Transitive ?
numbers of triangles = number of tape things,
number of tape things = number of geese
So number of triangles = number geese.

More important than this formal definition is understandimg how these classes allow us to generalize results.

Which of the following are equivalent in cardinality of sets?

****   ^^^^^^ $$$ ++++   %%%%%%%%

$$$$$$ ####

This means if we have a result about ****, we can apply this result (in context) to ####

We can do the same with modulo 4, put the following into equivalence classes

4 8 81 34 17 21 16 32 33 100 102

4 8 16 32 100 0

81 17 21 33 1

34 102 2

Easy Peasy right?

Take it up a notch

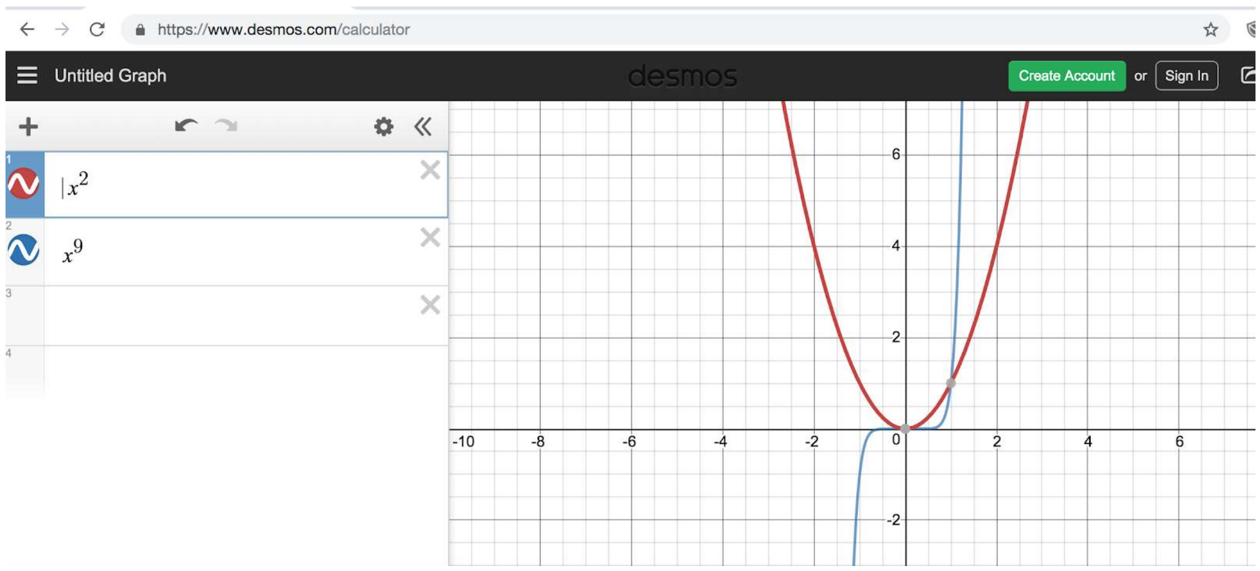
Let’s apply this to the idea of complexity - super important concept in computer science.

We represent the runtimes of algorithms with functions.

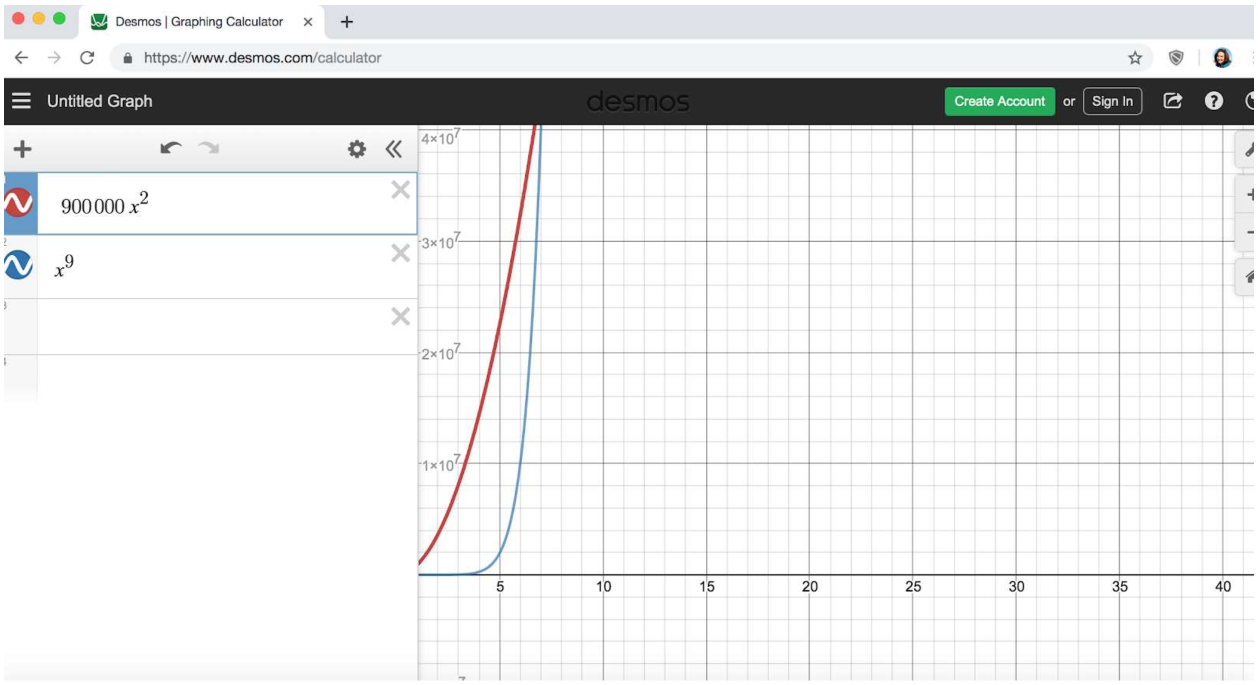
With x (or n) being numbers executions, and f(x) or (y) being time.

So if algorithm Red has complexity x^2 and algorithm Blue has complexity x^9, we consider Red to be a “faster” algorithm

After 1 iteration, Red is clearly faster, meaning as the number of runs (x-axis) goes on, the time (y axis) stays smaller than Blue.



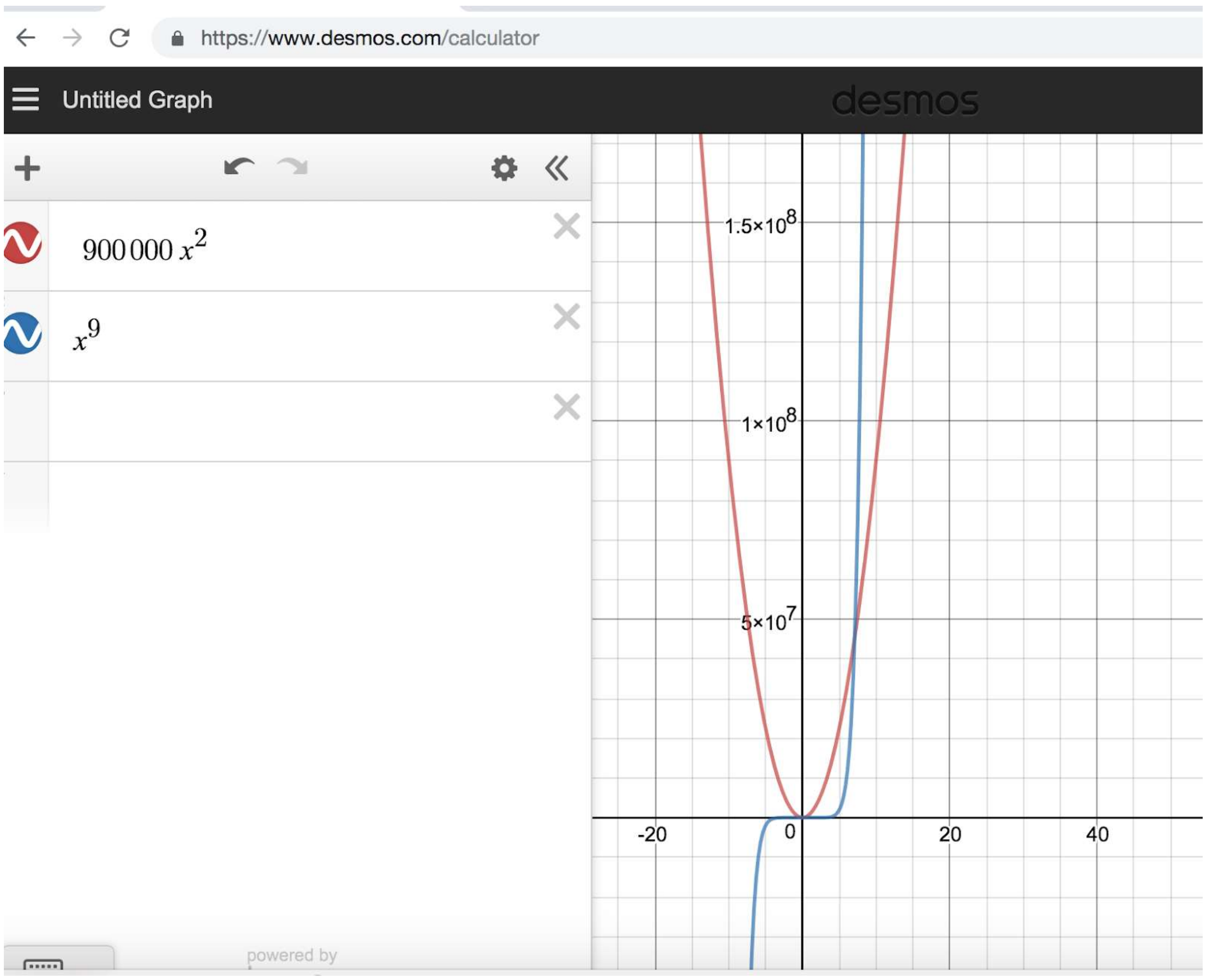
But what if we have Red at   900,000 x^2 and Blue is x^9 ? Now it looks like B is faster.



$900000x^2$

x^9

But looking out farther, we see Red really is better.



What is happening is that NO MATTER HOW BIG a coefficient we put in front of x^2, in the long run x^9, will always be faster. In the future, you will study this concept in a more formal way.

So how do we make sense of and compare algorithms when there are so many variations?

It turns out that the formal concept (called Big Theta) is an equivalence relation we can apply to all polynomials. See page 216 for the formal THM.

Informally, consider the set of ALL polynomials. Let's define an equivalence relation the set as follows.

The complexity of any polynomial is the x thing with the biggest exponent.

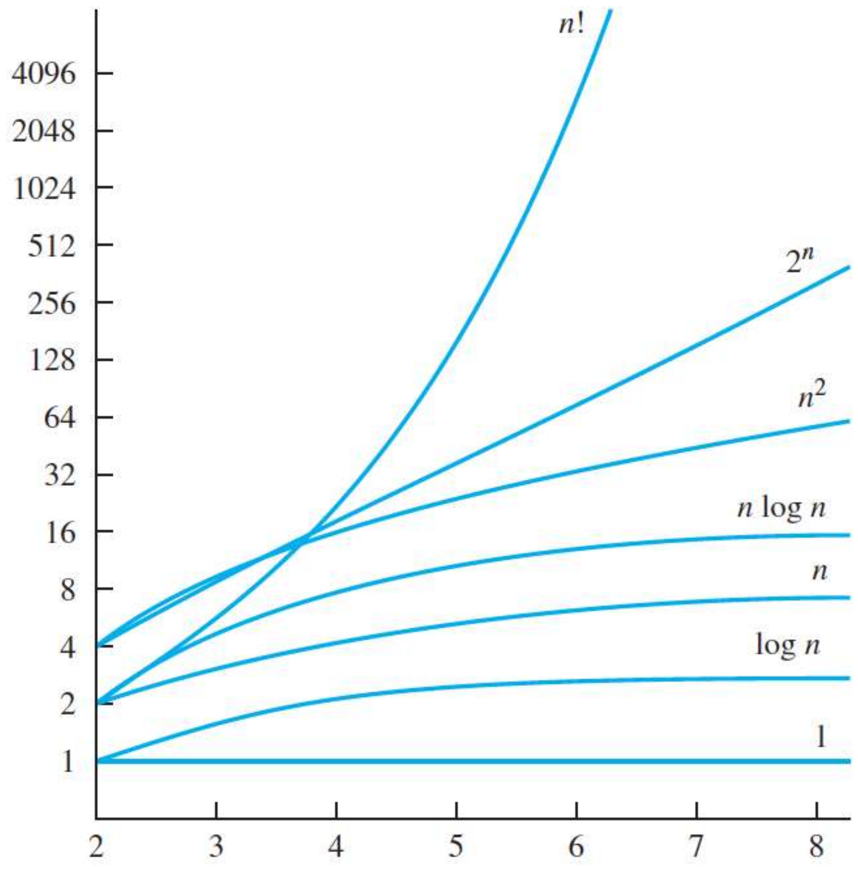
5 x^3 + 13 x^2 + 133 -------> x^3

17 x^5 + 27 x^3 + x -------> x^5

This lets us divide all polynomials into equivalence classes.

Now more broadly, in complexity we seek to categorize all runtimes into simple complexities, and in fact the polynomial complexity just gets rolled up into one big class. In the example below, they just use the example of n^2.

So all complexity seks to do is find a way to fundamentally categorize any function so we can know if we are making meaningful improvements.


$n!$
2^n
n^2
$n \log n$
n
$\log n$