Big O notation and comparison of functions

The functions that we will study in order are:

Constant $y=n$ $y=c$ $y=5$

Linear $y=x$

Quadratic $y=x^2$

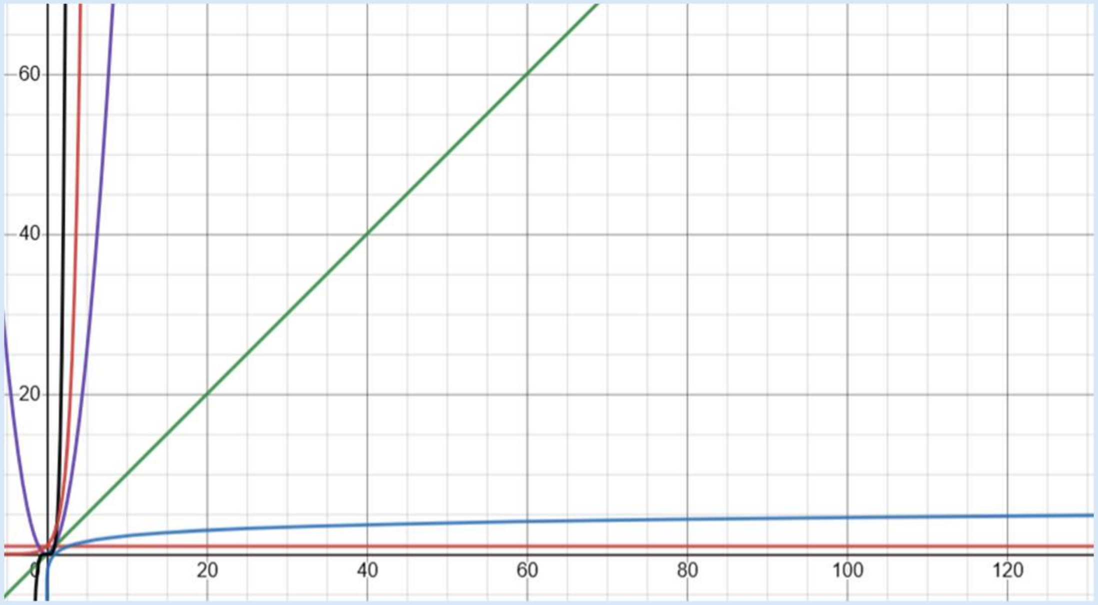
Polynomial $y=x^p$

Exponential $y=e^x$ $e^x=a$

Logarithmic $y=\ln a$

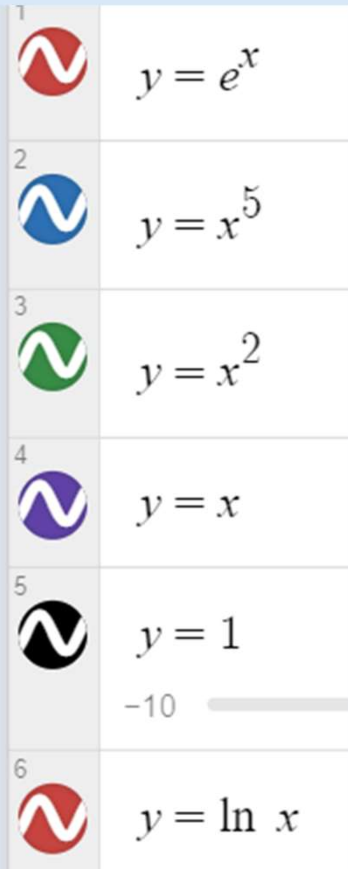
Big O notation is used in computer science to talk about the relative rates at which functions grow in relation to really big inputs. It can also be used to talk about run time or space requirements in relation to the size of the inputs. The O stands for order of the function.

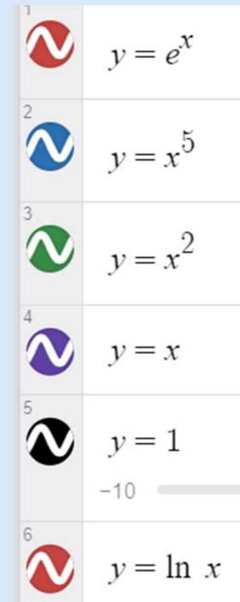
We are looking at how the functions grow for big numbers in relation to each other.



[Big O notation - Wikipedia]

We are looking at how the functions grow for big numbers in relation to each other.





Big O notation and comparison of functions from [Microsoft Word - Big O notation.doc (mit.edu)]

Big O notation uses a capital O and then has n as the input for a function that is in the parentheses.

notation	**name**
$O(1)$	constant
$O(\log(n))$	logarithmic
$O((\log(n))^c)$	polylogarithmic
$O(n)$	linear
$O(n^2)$	quadratic
$O(n^c)$	polynomial
$O(c^n)$	exponential

Big O notation uses a capital O and then has n as the input for a function that is in the parentheses.

O(c) is a constant function and doesn't grow. O(1) is also used.

O(log n) or O(ln n) is a logarithmic function. This is the inverse of the ln.

O(n) is a linear function. Of course, a steeper slope will grow faster, but we are just comparing to other functions and the steepest linear function doesn't grow as fast as a quadratic for big numbers.

O(n^2) is quadratic and grows faster than linear functions for big number.

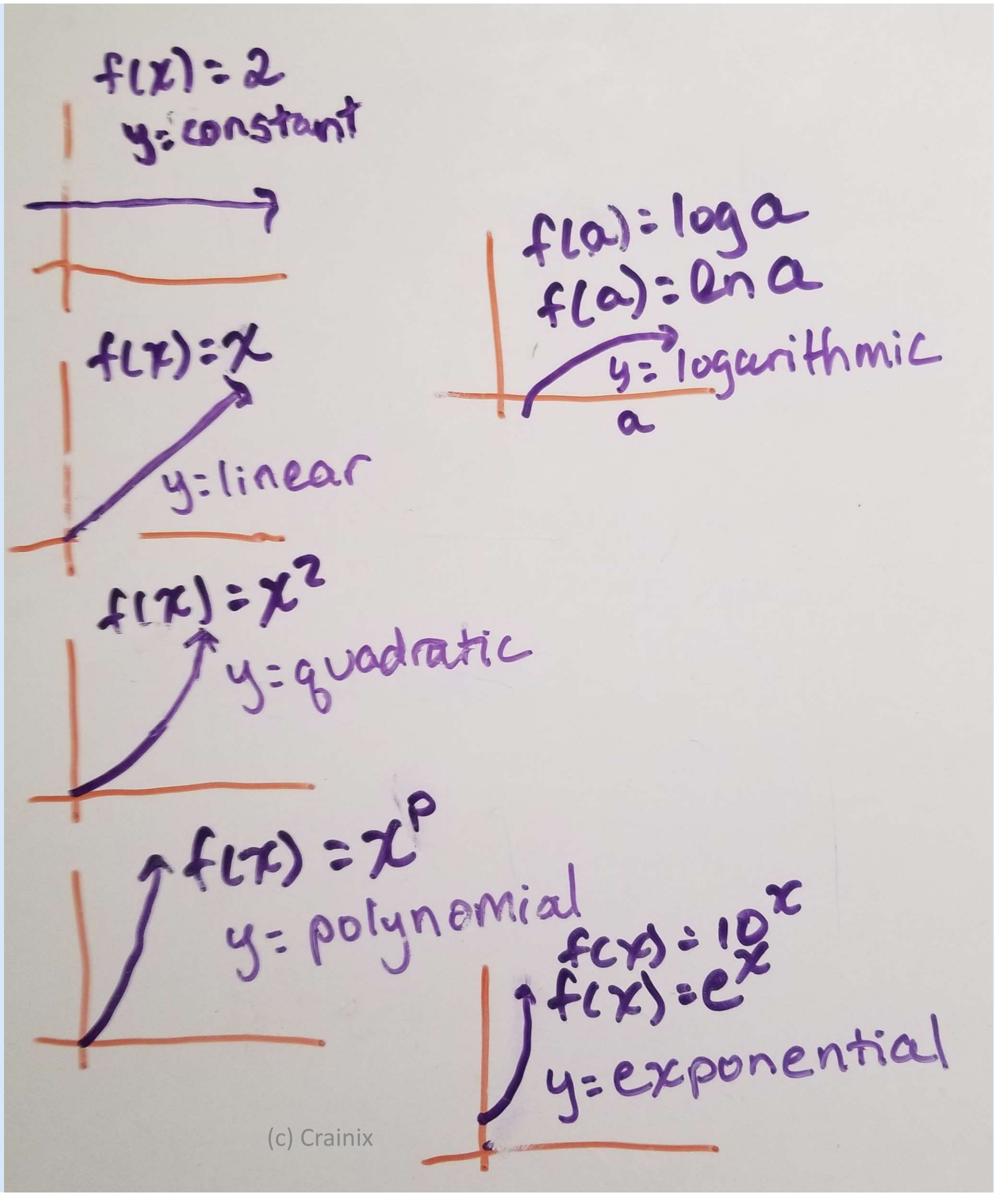
O(n^c) is polynomial and grows faster than lower powered polynomial functions for big number.

O(c^n) or O (e^n) is exponential and grows faster than the other functions.

From Wikipedia: Big O notation - Wikipedia

Notation	Name	Example
$O(1)$	constant	Determining if a binary number is even or odd; Calculating $(-1)^n$; Using a constant-size lookup table
$O(\log \log n)$	double logarithmic	Number of comparisons spent finding an item using interpolation search in a sorted array of uniformly distributed values
$O(\log n)$	logarithmic	Finding an item in a sorted array with a binary search or a balanced search tree as well as all operations in a Binomial heap
$O((\log n)^c)$ $c > 1$	polylogarithmic	Matrix chain ordering can be solved in polylogarithmic time on a parallel random-access machine.
$O(n^c)$ $0 < c < 1$	fractional power	Searching in a k-d tree
$O(n)$	linear	Finding an item in an unsorted list or in an unsorted array; adding two *n*-bit integers by ripple carry
$O(n \log^* n)$	*n* log-star *n*	Performing triangulation of a simple polygon using Seidel's algorithm, or the union–find algorithm. Note that $\log^*(n) = \begin{cases} 0, & \text{if } n \leq 1 \\ 1 + \log^*(\log n), & \text{if } n > 1 \end{cases}$
$O(n \log n) = O(\log n!)$	linearithmic, loglinear, quasilinear, or "*n* log *n*"	Performing a fast Fourier transform; Fastest possible comparison sort; heapsort and merge sort
$O(n^2)$	quadratic	Multiplying two *n*-digit numbers by a simple algorithm; simple sorting algorithms, such as bubble sort, selection sort and insertion sort; (worst case) bound on some usually faster sorting algorithms such as quicksort, Shellsort, and tree sort
$O(n^c)$	polynomial or algebraic	Tree-adjoining grammar parsing; maximum matching for bipartite graphs; finding the determinant with LU decomposition
$L_n[\alpha, c] = e^{(c+o(1))(\ln n)^\alpha (\ln \ln n)^{1-\alpha}}$ $0 < \alpha < 1$	L-notation or sub-exponential	Factoring a number using the quadratic sieve or number field sieve
$O(c^n)$ $c > 1$	exponential	Finding the (exact) solution to the travelling salesman problem using dynamic programming; determining if two logical statements are equivalent using brute-force search
$O(n!)$	factorial	Solving the travelling salesman problem via brute-force search; generating all unrestricted permutations of a poset; finding the determinant with Laplace expansion; enumerating all partitions of a set

Graphs of the functions:



n! is not a function but a notation and it comes up in growth for combinatorics, and growth of computer algorithms.

$$n!$$

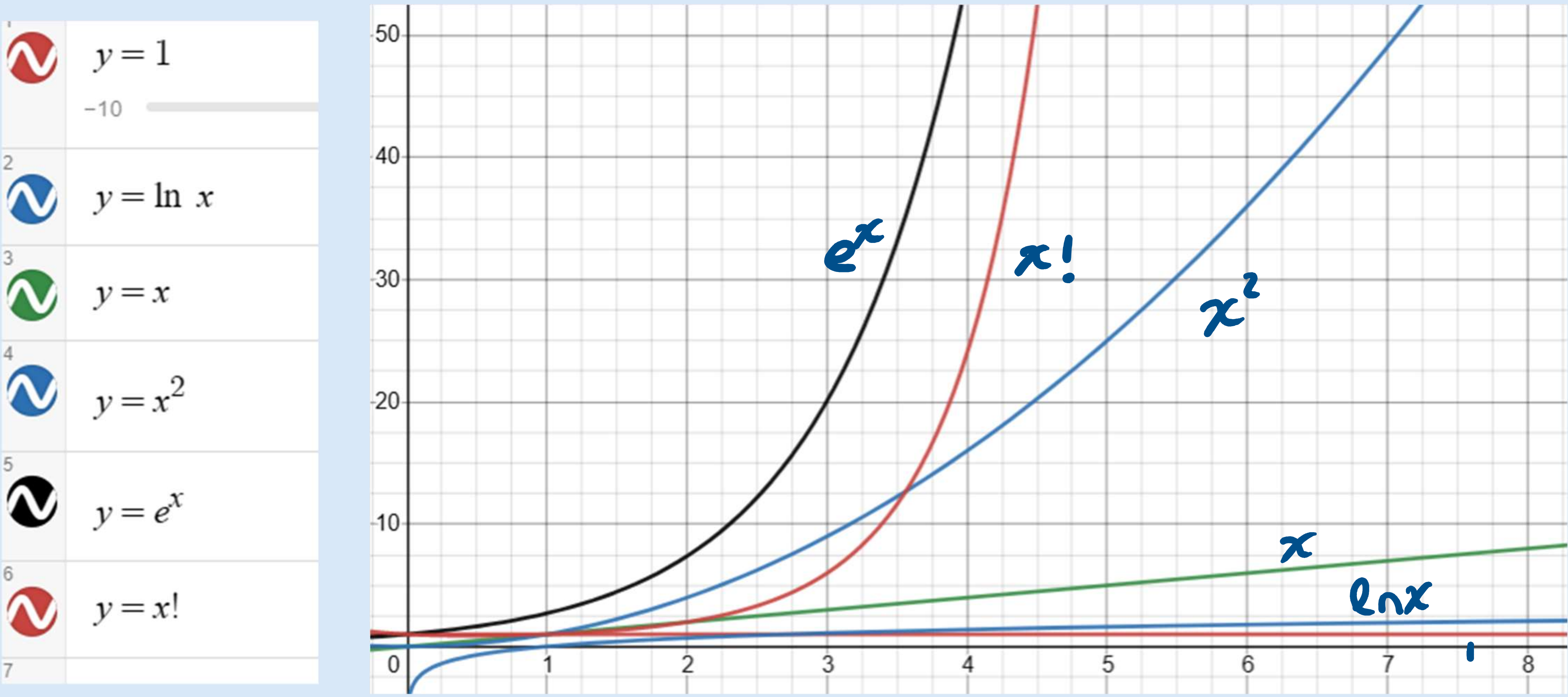
$$1! = 1$$

$$2! = 2 \cdot 1$$

$$3! = 3 \cdot 2 \cdot 1$$

$$6! = 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$

In Desmos, we can see how it graphs out. The n! doesn't grow bigger than e^x until later, but it does grow more for big numbers. Big O is used for when x gets really big or approaches infinity.



Zooming out in Desmos, you can see when n! overtakes the exponential.