Big-Oh

Big-Oh Notation, where *n* refers to the size of the problem (eg, n is the length of the array)

$O(1)$ = "Constant Time" – runtime does not depend on n

$O(\log n)$ = "Logarithmic Time" – runtime is proportional to log n
Clue: Every time you double the problem size, runtime grows by a constant

$O(n)$ = "Linear Time" – runtime is proportional to n
Clue: Every time you double the problem size, time doubles

$O(n^2)$ = "Quadratic Time" – runtime is proportional n^2
Clue: A linear time operation applied a linear number of times

$O(2^n)$ = "Exponential Time" – runtime is proportional 2^n
Clue: Add one to the problem size, runtime doubles

Suppose we know that the running time for an algorithm follows a function Time(n) – time versus the input size n.

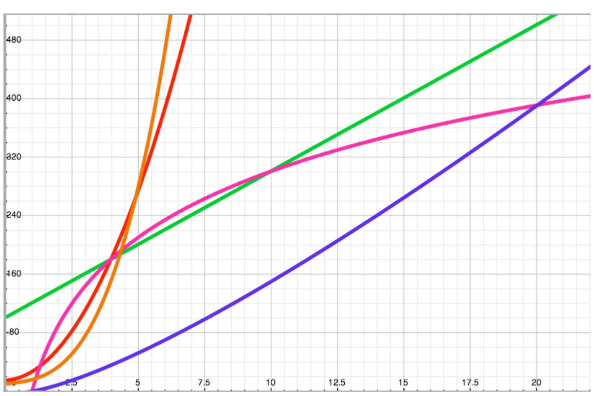
$\text{Time}(n) = 20\,n + 100 = O(n)$

$\text{Time}(n) = 10\,n^2 + 2x + 15 = O(n^2)$

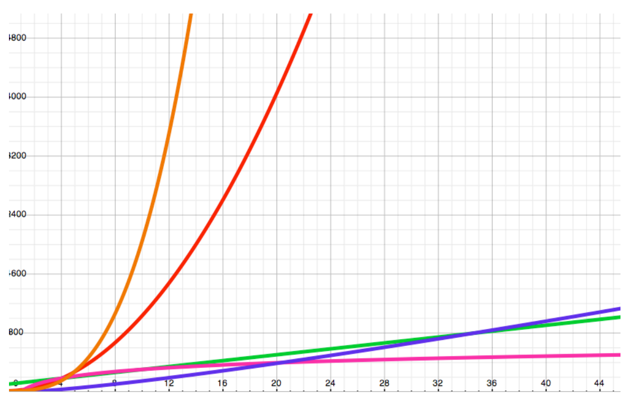
$\text{Time}(n) = 300 \log(n) + 25 = O(\log n)$

$\text{Time}(n) = 2\,n^3 + 4n + 10 = O(n^3)$

$\text{Time}(n) = 15\,n * \log(n) = O(n \log n)$



Small n



Larger n

Visitor: How old are those dinosaur bones?
Curator: 25 million and 38 years.
Visitor: Wow, how do you know that?
Curator: When I came here they told me the bones were 25 million years old and I've been here 38 years.

To wit:

For sufficiently large n, the value of a function is determined by its dominant term.
Big-Oh captures the most dominant term in a function.
It ignores multiplicative constants.
It represents the growth rate of the function.

Rule: For operation A follow by operation B, take the MAXIMUM Big-Oh for the overall Big-Oh.

Time(n) = O(n) + O(n) = O(n)
O(n) operation followed by another O(n) operation

Time(n) = O(n) + O(n^2) = O(n^2)
O(n) operation followed by a O(n^2) operation

Time(n) = O(n) + O(log n) = O(n)

Rule: Operation A that executes operation B, take the PRODUCT of Big-Oh

Time(n) = O(n) * O(n) = O(n^2)
O(n) operation that executes an O(n) operation
(eg, two nested loops, eg, a loop that calls an O(n) operation)

Time(n) = O(n) * O(log n) = O(n log n)
O(n) operation repeated O(log n) times

Time(n) = O(n) * O(1) = O(n)

n	Constant	Logarithmic	Linear	Quadratic
1000	10 ns	100 ns	10 μs	10 ms
1 million	10 ns	200 ns	10 ms	3 hours
1 billion	10 ns	300 ns	10 sec	300 years

Subset Sum: Find the sum of every possible subset of a set of numbers and see if any subset sums to a target value.

How many possible subsets are there of n elements? 2^n (one for each binary string of length n). Thus, it is an O(n^2) algorithm, "exponential" time.

How slow is that? Really, really slow. No one knows of a faster algorithm.

n	Logarithmic	Linear	Quadratic	Exponential
10	< 1 ms	< 1 ms	< 1 ms	10 ms
20	< 1 ms	< 1 ms	< 1 ms	10 sec
30	< 1 ms	< 1 ms	< 1 ms	3 hours
40	< 1 ms	< 1 ms	< 1 ms	130 days
50	< 1 ms	< 1 ms	< 1 ms	400 years
60	< 1 ms	< 1 ms	< 1 ms	400,000 yrs
70	< 1 ms	< 1 ms	< 1 ms	400 million years

What does Big-Oh tell you?

- It does **not** tell you the numerical running time of algorithm for a particular input or for small n.
- It tells you something about the **rate of growth** as the size of the input increases.
- At some point O(n) algorithm will be faster than an O(n^2) algorithm, always.
- Furthermore, as the input size grows, the O(n) algorithm will get increasingly faster then an O(n^2) algorithm.
- But it will **not** tell you for what values of n the O(n) algorithm is faster than the O(n^2) algorithm.
- Similarly, an O(n log n) algorithm will get increasingly faster than O(n^2) algorithm.
