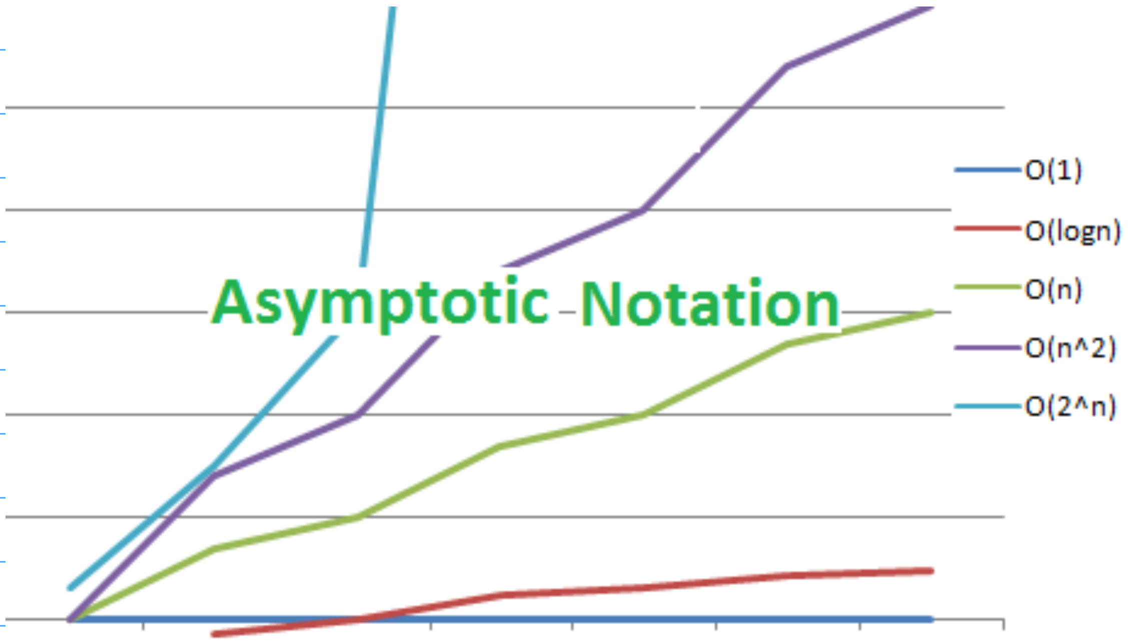
Running time of an algorithm is measured as the function of its input size.

Longer inputs → more time



$\log n$	n	n^2	2^n
0	1	1	2
0.301	2	4	4
0.4771	3	9	8
0.6021	4	16	16
0.6989	5	25	32
0.7781	6	36	64
0.845	7	49	128

Big-Oh notation

example of asymptotic notation

Its a way of comparing two functions just on their growth rate and not on their details.

log n is Big-Oh to 2^n function

log n grows much slowly than 2^n fun

Big-Oh $\longrightarrow$ <=

less than equal to

Big-omega (converse of Big-Oh)

$\longrightarrow$ >=

Greater than equal to

2^n is big-omega of log n

Technically

$O(f(n)$ is a set

Order $f(n)$ for any function $f(n)$ is actually the set of all functions whose growth rate is the same or less than the growth rate of $f(n)$

$O(n^2)$ is set of all functions that do not grow faster than n^2

$2n^2$ grows as fast as n^2
so, its $O(n^2)$

$2n^2$

$O(n^2)$

$100n^2+n+1$

n and 1 grows more slowly than $100n^2$

n

$2n^2 \in O(n^2)$
$100n^2+n+1 \in O(n^2))$
$n \in O(n^2)$

omega

Some elements of $\Omega(n^2)$

$\geq$

$\Omega(n^2)$
- $2n^2$
- $\frac{n^2}{1000} - n$
- n^3

$2n^2 \in \Omega(n^2)$

$\frac{n^2}{1000} - n \in \Omega(n^2)$

$n^3 \in \Omega(n^2)$

Insertion sort has a runtime of $O(n^2)$

Complexity

More on Insertion Sort...

```
insertion-sort A:
    for i <- 1 to length(A)
        j <- i
        while j > 0 and A[j-1] > A[j]:
            swap A[j] and A[j-1]
            j <- j-1
```

How does insertion sort perform on an already-sorted array? | 1 | 2 | 3 | 4 |

- time for inner loop?
- Each iteration requires no swaps! (constant time to check the first element)
- $\sum_{i=1}^{n} 1 = n \in O(n)$

In insertion Sort $\rightarrow O(n^2)$
But when its in sorted order its
$\rightarrow O(n)$

There are many inputs of the same length
Algorithm might be very fast on some of these inputs and very slow on some of the other inputs

We always take pessimistic view that input will not be favorable to us.

We have to have an algorithm that that does well even on worst case input.

Divide and conquer

you have a problem, you devide it into subproblems.
Do the conquer step where you solve each of the subproblems

Then combine the solutions to solve the whole problem.