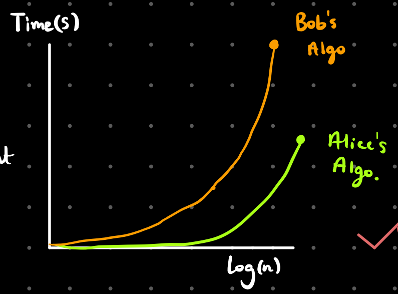
Big O Notation

- $f = O(g)$ if there is a constant $c > 0$ such that $f(n) \leq c \cdot g(n)$
  ↳ But what does this really mean?

- Measuring efficiency of an algorithm is very important

Using sample data input testing different n inputs.



→ Visual Graph to determine efficiency between 2 algos.

→ Not practical to gather time data for each algorithm!

Machine Dependence - different computers may compute different results

We want to measure machine independence
↳ count operations in algorithm and see how it changes as input grows
(care about worst case scenario)

Triple for() loop → n^3
*uses actual #'s given n
Double for() loop → n^2

→ Issues, takes too long to come up with

We need to generalize the algorithm for any n!

Bob → $(n+1)^3 \approx n^3$
Alice → $(n+1)^2 \approx n^2$

Running Time! (Drop constants since we only care about when n grows very large! Only biggest term matters)

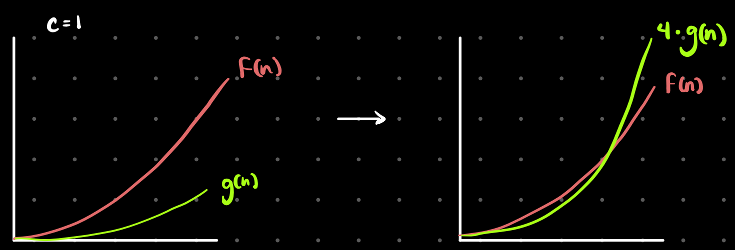
$O(n^2)$ is better than $O(n^3)$

Big O Common Misconceptions

$f = O(g)$ if there is a constant $c > 0$ such that $f(n) \le c \cdot g(n)$ for large n

↳ $f(n)$ "grows no faster" than $g(n)$

Ex. $f(n) = 3n^2 + 5n + 4 \rightarrow O(n^2) \longrightarrow g(n) = n^2$



Classes of Running Time!

Constant – $O(1)$

Logarithmic – $O(\log(N))$

Linear – $O(N)$

Linearithmic – $O(N * \log(N))$

Polynomial – $O(N^2)$, $O(N^3)$, etc.

Exponential – $O(2^N)$, $O(3^N)$, etc.

Identifying Algo's Running Time

① Understand how algo works
- purpose
- input(s)
- outputs

② Identify basic unit of algorithm to count
- print statements
- iterations/assignment statements
- recursive calls

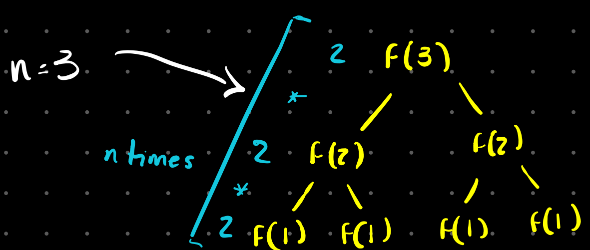
- focus on WORST case

③ Map growth of count from Step 2 to appropiote
- is growth constant?
- exponential?
- linear?
- logarithmic

```
def f(n):
    if n==1:
        return 1

    else:
        return f(n-1) + f(n-1)
```

- Base Case n=1
- Sum of two recursive calls to f(n-1)



As n grows, multiply 2, n times over (to take recursive calls into account)

$O(2^n)$ Time Complexity