# Algorithm Analysis / Big-Oh

# Why analyze algorithms?

- CPU time (effort) is a valuable resource
- There can be multiple different algorithms and implementations to solve a problem
- Want to make sure we choose an efficient approach
  - We'll focus on efficiency with respect to operations performed by CPU
  - Can also choose / analyze with respect to others such as space on computer used
  - Often tradeoff between these

# Algorithm

- Not the actual code (aka "implementation")
- The steps you take to solve the problem
- Analysis should happen:
    - at the algorithm stage
    - before you start coding

# Complexity

- How does the time depend on the problem size?
    - Use the phrase "time", but we don't really care about exact execution time
    - Look at growth function that says how time behaves
    - Actually only care about **asymptotic complexity**
        - Long term behavior as n increases
        - Based on dominant term
- Need to defined problem size (n)
    - Ex: # elements in array

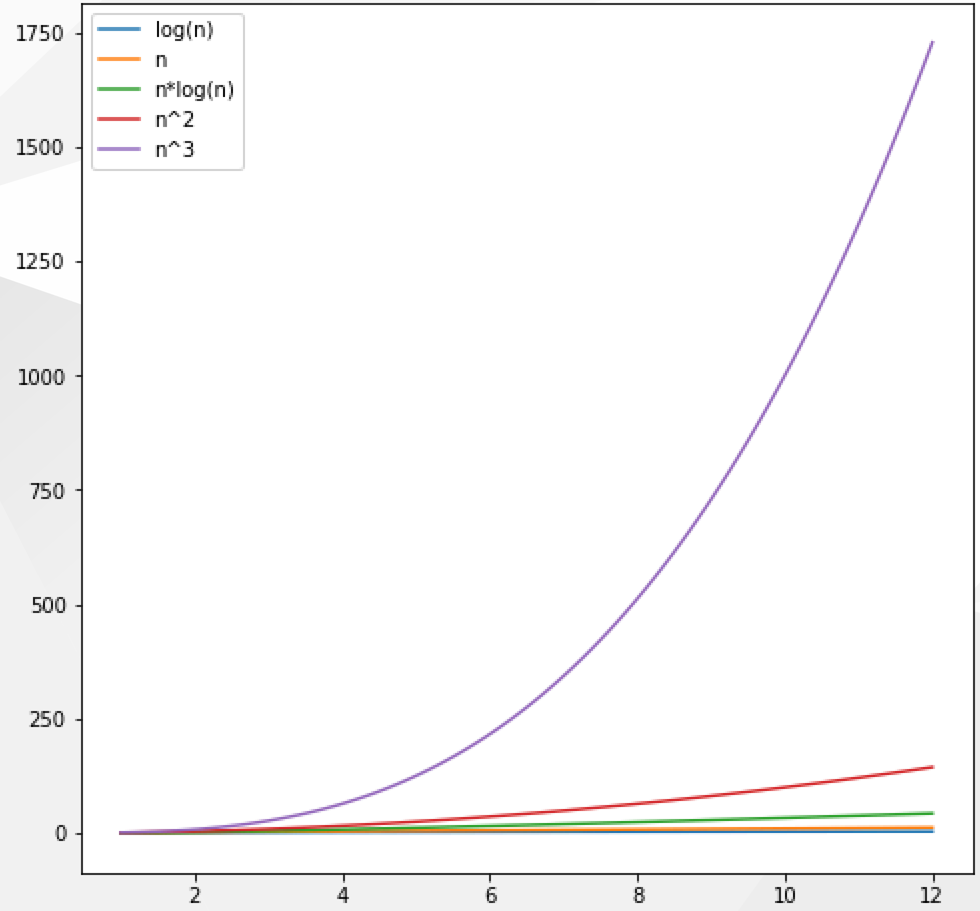
# Complexity

- Aka **order** of algorithm
- Big-Oh notation
  - O(n^2) -> behaves like n^2 as n increased
- Big-Oh formally:
  - $f(n) = O(g(n))$ if there exists constants $c$ and $m$ such that $f(n) < cg(n) \forall n \geq m$
- Basically, upper bound to growth function

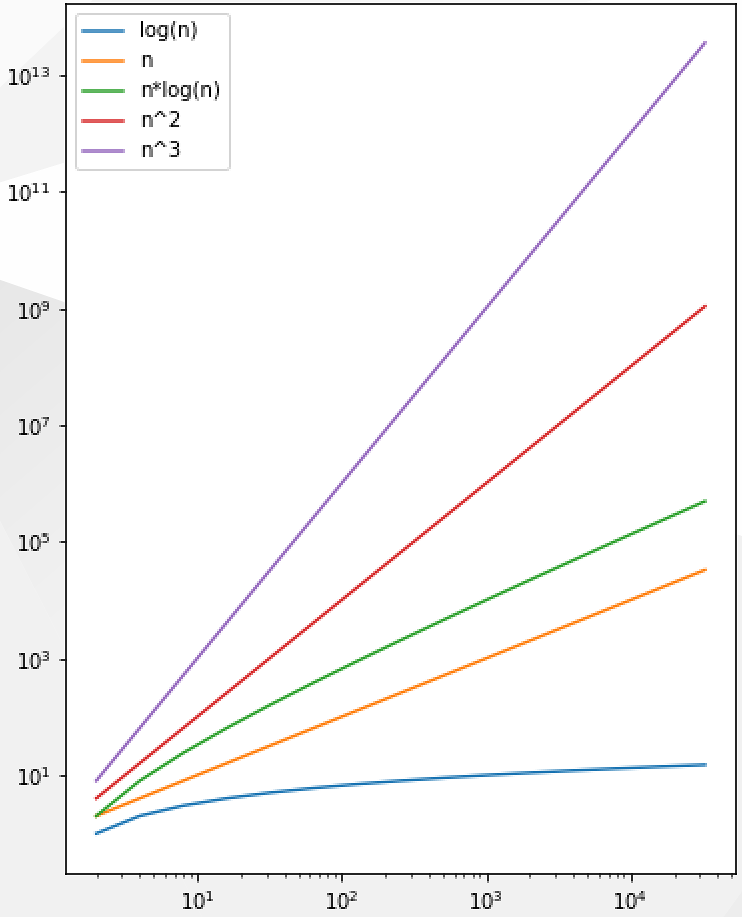
## Examples

- $2n^3 + 12n^2 - n + 4$
- $4n^2 + 15n$
- $n + n * \log(n) + 6$
- $\log(n) + 12$

# Complexity Visualizations



# Complexity Visualizations


log(n)
n
n*log(n)
n^2
n^3
$10^{13}$
$10^{11}$
$10^{9}$
$10^{7}$
$10^{5}$
$10^{3}$
$10^{1}$
$10^{1}$
$10^{2}$
$10^{3}$
$10^{4}$