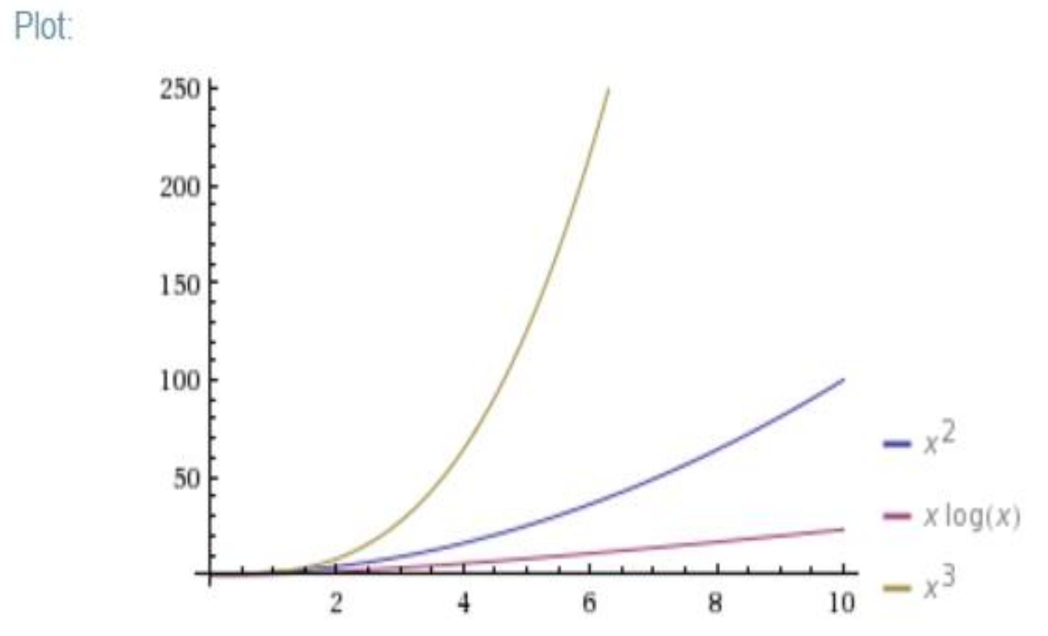
CS 325 – Asymptotic Analysis

Week 1 Part 2- Big-Oh, Omega, Theta

Problems vs Algorithms

- For any given problem there are potentially many different types of algorithms to solve it.
- Problem: Sorting a list of integers
- Algorithms: Insertion Sort, Merge Sort, Naive Sort
- Running time
  - Insertion Sort is $O(n^2)$
  - Merge Sort is $O(n\lg n)$
  - Naive Sort is $O(n^3)$



How do we compare algorithms?

We need to define a number of <u>objective measures</u>.

(1) Compare execution times?

Not good: times are specific to a particular computer and programming language!!

(2) Count the number of statements executed?

Not good: number of statements vary with the programming language as well as the style of the individual programmer.

Types of Analysis

- Worst case
  - Provides an upper bound on running time
  - An absolute **guarantee** that the algorithm would not run longer, no matter what the inputs are
- Best case
  - Provides a lower bound on running time
  - Input is the one for which the algorithm runs the fastest
- Average case = Expected Value
  - Provides a prediction about the running time
  - Assumes that the input is random

Asymptotic Analysis

To compare two algorithms with running times *f(n)* and *g(n)*, we need a **rough measure** that characterizes **how fast each function grows.**

- Running time of an algorithm as a function of input size n **for large *n***.
- Compare functions in the limit, that is, **asymptotically!** (i.e., for large values of n)
- Worst Case Analysis

Input Size

Express running time as a function of the input size *n* (i.e., f(n)).

- size of an array
- # of elements in a matrix
- # of bits in the binary representation of the input
- vertices and edges in a graph

Constant factors and domination

Suppose we have two algorithms with exact running times of:

Algorithm 1		Algorithm 2
$1{,}000{,}000 \cdot n$	versus	$2 \cdot n^2$

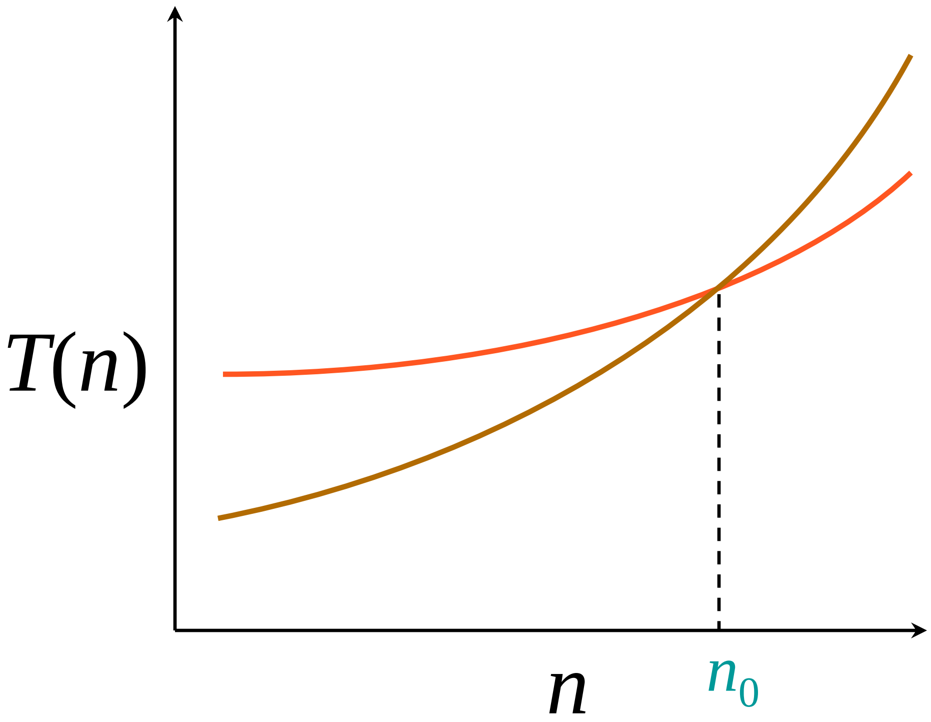
Is it reasonable to say that runtime of Algorithm 2 dominates (is worse) than Algorithm1?

NO for small values of n Algorithm 2 is better

Yes for large values of n

Asymptotic performance

When n gets large enough, an n^2 algorithm always "*beats*" a n^3 algorithm.

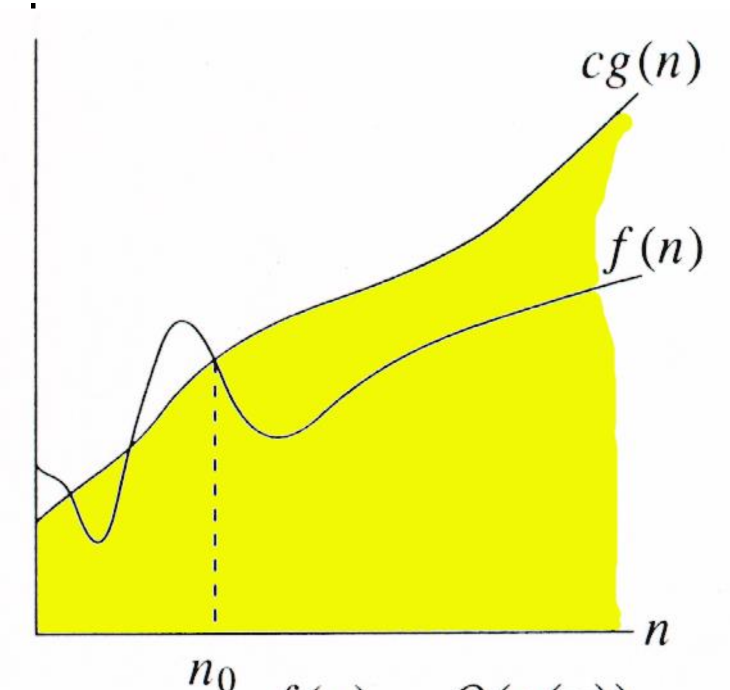


Asymptotic Notation

- Big-Oh - O notation: asymptotic "less than":
  - $f(n)=O(g(n))$ implies: $f(n)$ "$\leq$" $g(n)$
- Omega - Ω notation: asymptotic "greater than":
  - $f(n)= \Omega (g(n))$ implies: $f(n)$ "$\geq$" $g(n)$
- Theta - Θ notation: asymptotic "equality":
  - $f(n)= \Theta (g(n))$ implies: $f(n)$ "$=$" $g(n)$

O-notation

$$O(g(n)) = \{f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\} .$$



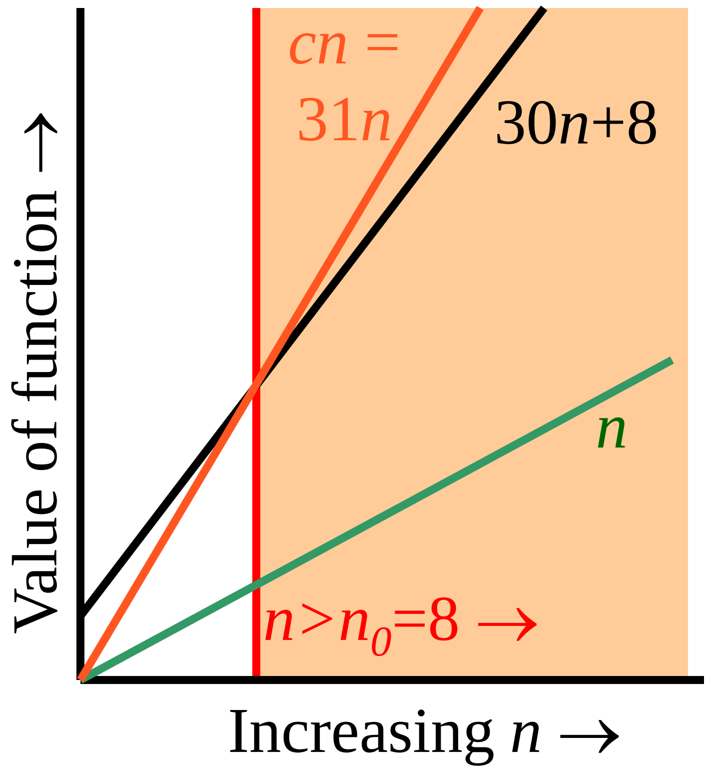
$f(n) = O(g(n))$

$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

Big-O example, graphically

- Note $30n+8$ isn't less than n *anywhere* ($n>0$).
- It isn't even less than $31n$ *everywhere*.
- But it *is* less than $31n$ <u>everywhere to the right of $n=8$</u>.

30*n*+8 is O(*n*)



Not Unique - Example

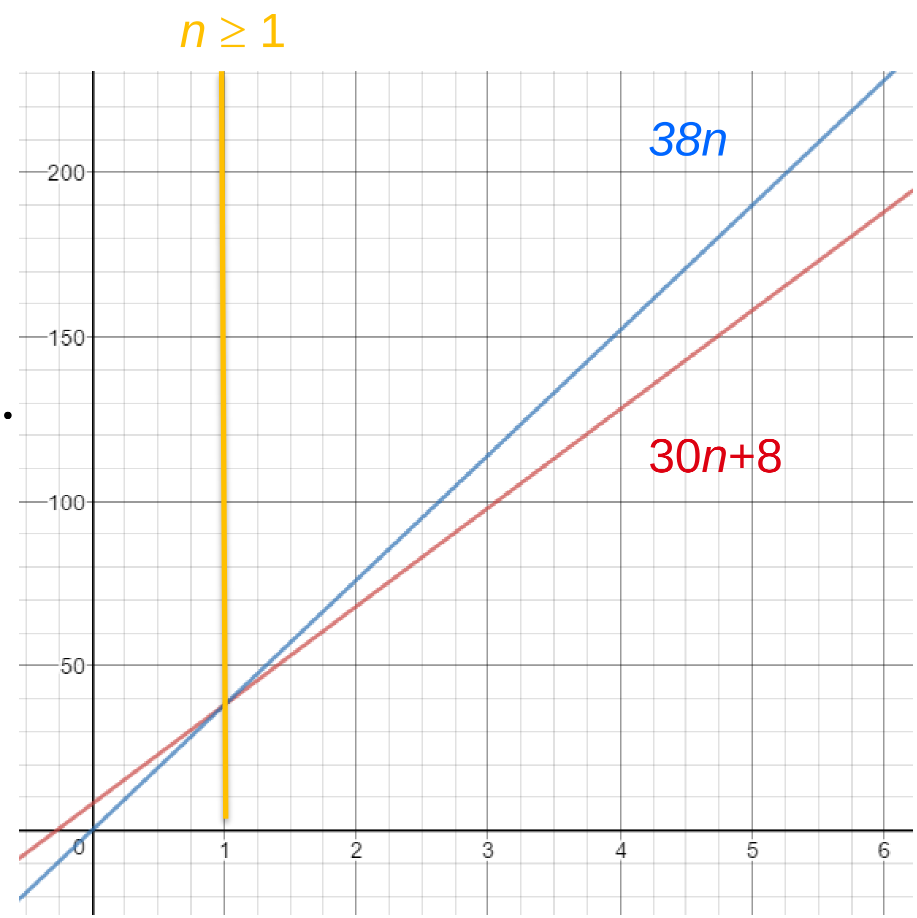
Show that $30n+8$ is $O(n)$.

Show $\exists c, n_0$: $30n+8 \leq cn$, $\forall n \geq n_0$.

Let $c=38$, $n_0=1$. Assume $n \geq n_0=1$.

Then $cn = 38n = 30n + 8n \geq 30n +$

$30n+8 \leq 38n$. when $n \geq 1$



A Simple Code Example

- Consider summing an array of n integers.

```
sum = 0;
for (i = 0; i < n; i++)
    sum += array[i];
return sum;
```

`sum = 0;` — Executes in constant time c_1 (independent of n)

`for (i = 0; i < n; i++) sum += array[i];` — Executes in $c_2 \cdot n$ time for for some constant c_2

`return sum;` — Executes in constant time c_3 (independent of n)

- Total running time: $c_1 + c_2 n + c_3$
  - But the constants c_1, c_2, c_3 depend on hardware, compiler, etc.
- What is the big-Oh runtime? (big-Oh ignores factors)

O(n) also known as <u>linear time</u>

A Simple Example

- Consider summing an array of *n* integers.

```
sum = 0;
for (i = 0; i < n; i++)
    sum += array[i];
return sum;
```

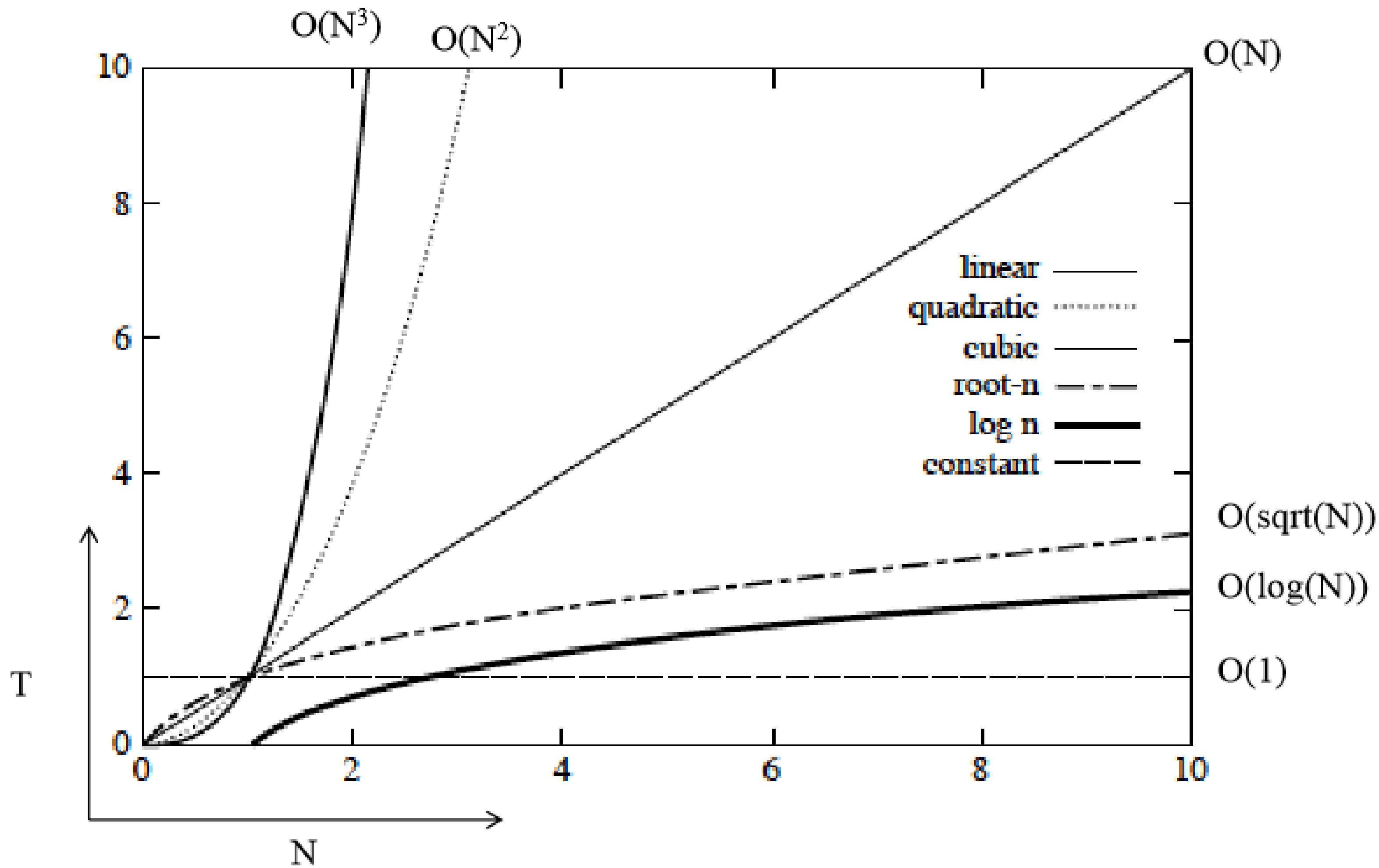
$\Big\} \; O(n)$

Non-Linear Times

- Consider the Insertion Sort Algorithm
  - Let *n* be the size of the input list to be sorted
  - Runtime is $O(n^2)$, also known as quadratic time.
- Suppose size doubles, what happens to execution time?

- It goes up by a factor of 4. Why?

Orders of Growth



Big Oh Classes

- Constant — $O(1)$
- Logarithmic — $O(\log(n))$
- Linear — $O(n)$
- Quadratic — $O(n^2)$
- Cubic — $O(n^3)$
- Polynominal — $O(n^k)$ for any $k>0$
- Exponential — $O(k^n)$, where $k>1$
- Factorial — $O(n!)$

Rank the following functions in increasing order of growth

$10n$, $\lg(2^n)$, 1000, $\mathrm{sqrt}(n)$, $3n^2$, $n!$, $\log n$, 3^n

A polynomial of degree k is $O(n^k)$

Recall: $f(n)$ is $O(g(n))$ if there exist positive constants c and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$

Proof:

Suppose $f(n) = b_k n^k + b_{k-1} n^{k-1} + \ldots + b_1 n + b_0$

Let $a_i = | b_i |$

$$f(n) \leq a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$$

$$f(n) \leq n^k \left(a_k + a_{k-1} \frac{n^{k-1}}{n^k} + \ldots + a_1 \frac{n^1}{n^k} + a_0 \frac{1}{n^k} \right)$$

$$f(n) \leq \quad n^k \sum a_i \frac{n^i}{n^k} \leq n^k \sum a_i$$

$$let \;\; c = \sum a_i$$

$$f(n) \leq \quad cn^k \;\; \text{for n} \geq 1$$

Therefore all polynomial functions $f(n)$ of degree k are $O(n^k)$.

What does this mean in practice?

```
for(s = 0; s < N; s++)
        sum = sum + 1;

for(i = 0; i < N; i++)
    for(j =0; j < N; j++)
        printf( …)
```

$O(n)$

$O(n^2)$

Total = $O(n) + O(n^2) = O(n + n^2) = O(n^2)$

Code Example

```
int isPrime (int n) {
  for (int i = 2; i * i <= n; i++) {
    if (0 == n % i) return 0;
  }
  return 1; /* 1 is true */
}
```

What is the "Big-Oh" running time in terms of n?

$$O(\sqrt{n})$$

Trouble with Big-Oh

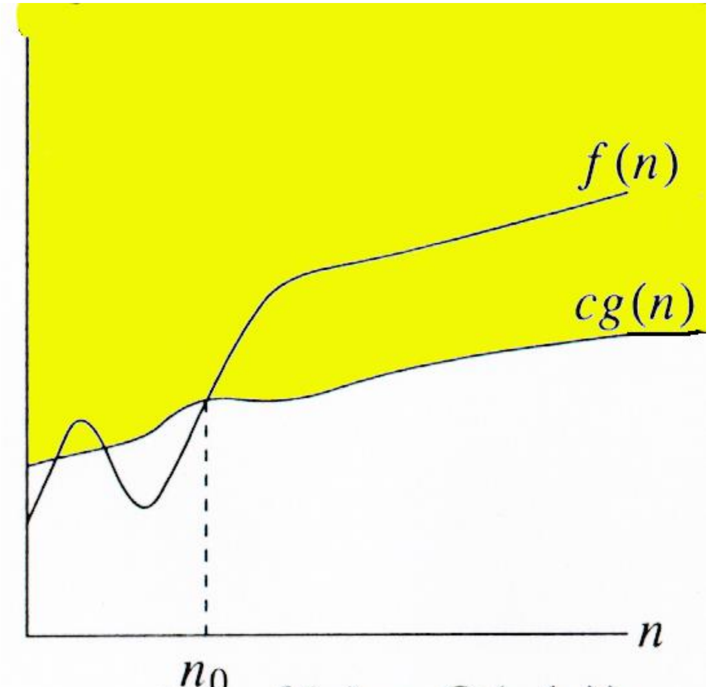
Just an upper bound. Factually true but practically meaningless.

- $3n$ is $O(n^2)$
- $3n$ is $O(n^4)$
- $3n$ is $O(n)$

Many times only Big-Oh is reported but it is assumed a “tight” upper bound.

Omega Ω -notation

$$\Omega(g(n)) = \{f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\}\ .$$



$f(n) = \Omega(g(n))$

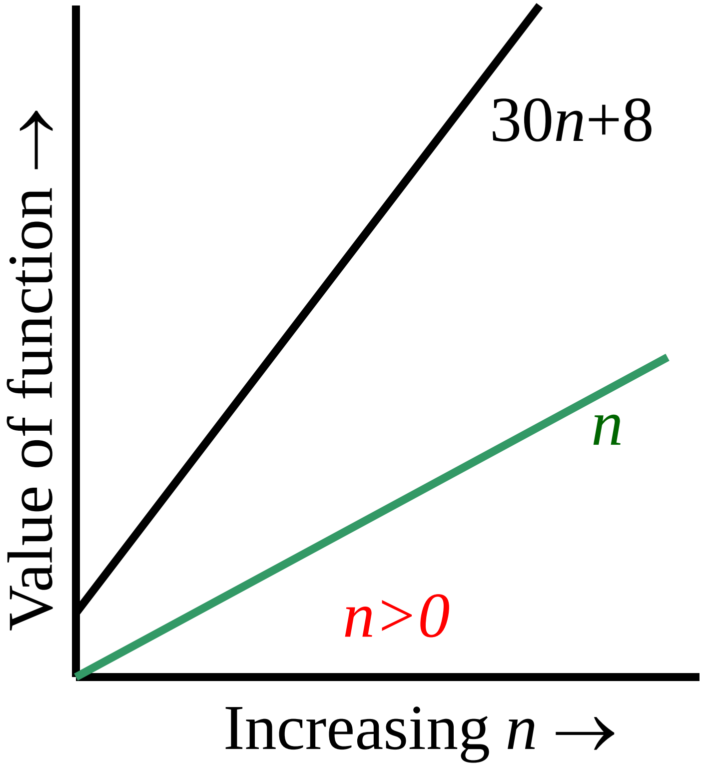
$\Omega(g(n))$ is the set of functions with larger or same order of growth as $g(n)$

$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

Omega Graphically

- Note $30n+8$ isn't less than n *anywhere* ($n>0$).

$30n+8$ is $\Omega(n)$



Examples

$5n^2 = \Omega(n)$

$\exists$ c, n_0 such that: $0 \le cn \le 5n^2$

$\Rightarrow cn \le 5n^2$

$\Rightarrow c = 5$ and $n_0 = 1$

$5n^2+10= \Omega(n^2)$

$\exists$ c, n_0 such that: $0 \le cn^2 \le 5n^2+10$

$\Rightarrow 5n^2c \le 5n^2 +10$

$\Rightarrow c = 1$ and $n_0 = 1$

Property of Big-Oh and Omega

If $f(n) = O(g(n))$ then $g(n) = \Omega(f(n))$

By definition of Big-Oh

$$f(n) \le cg(n) \text{ for all } n \geq n_0 \text{ for some } n_0, c > 0.$$

Dividing by c yields

$$\frac{1}{c} f(n) \le g(n) \text{ or } c_2 f(n) \le g(n) \text{ where } c_2 = \frac{1}{c} \geq 0$$

If we use the same n_0. this implies that $g(n) = \Omega(f(n))$.

A non-negative polynomial of degree k is $\Omega(n^k)$

Proof:

Suppose $f(n) = b_k n^k + b_{k-1} n^{k-1} + \ldots + b_1 n + b_0$

$$f(n) = n^k \left(b_k + b_{k-1}\frac{n^{k-1}}{n^k} + \ldots + b_1 \frac{n^1}{n^k} + b_0 \frac{1}{n^k} \right)$$

$$f(n) = n^k \left(b_k + \frac{b_{k-1}}{n^1} + \ldots + \frac{b_1}{n^{k-1}} + \frac{b_0}{n^k} \right)$$

For large n's the fractions go to zero, so if we set n_0 large enough we can ignore all terms except b_k . Thus a value for n_0 must exist.

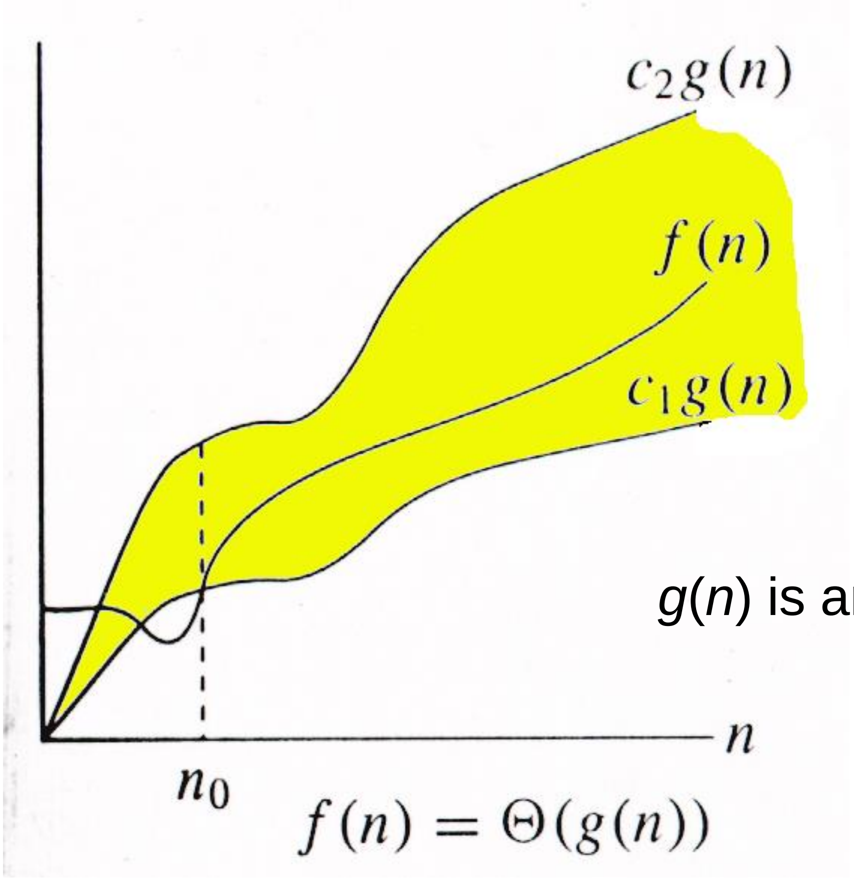
$$let\ \ c = \frac{\mathrm{b_k}}{2} \qquad\qquad cn^k \leq f(\mathrm{n}) \text{ for } \mathrm{n} \geq \mathrm{n_0}$$

$$\frac{\mathrm{b_k}}{2} n^k \leq \mathrm{b_k} n^k \text{ for } \mathrm{n} \geq \mathrm{n_0}$$

Therefore all polynomial functions $f(n)$ of degree k are $\Omega(n^k)$.

Θ-notation

$$\Theta(g(n)) = \{ f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \} .$$



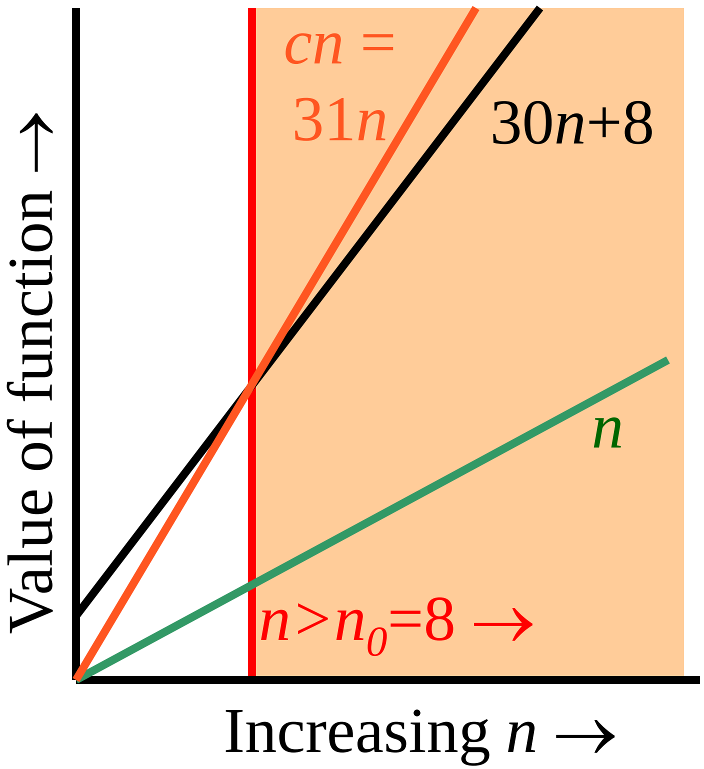
Θ(*g*(n)) is the set of functions with the same order of growth as *g*(n)

g(*n*) is an *asymptotically tight bound* for *f*(*n*).

Big-Theta example, graphically

- Note $30n+8$ isn't less than n *anywhere* ($n>0$).
- It isn't even less than $31n$ *everywhere*.
- But it *is* less than $31n$ <u>everywhere to the right of $n=8$</u>.

$$30n+8 \in \Theta(n)$$



A non-negative polynomial of degree *k* is $\Theta(n^k)$

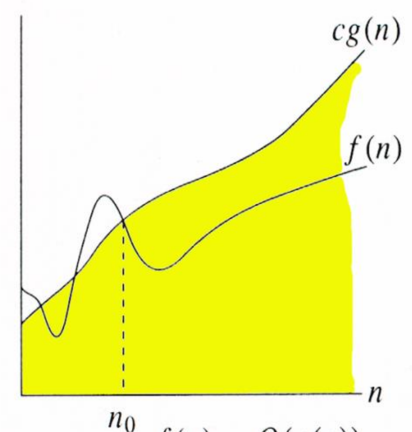
Proof: From previous Big-Oh and Omega

$f(n) = b_k n^k + b_{k-1} n^{k-1} + \ldots + b_1 n + b_0$ *is* $\Theta(n^k)$

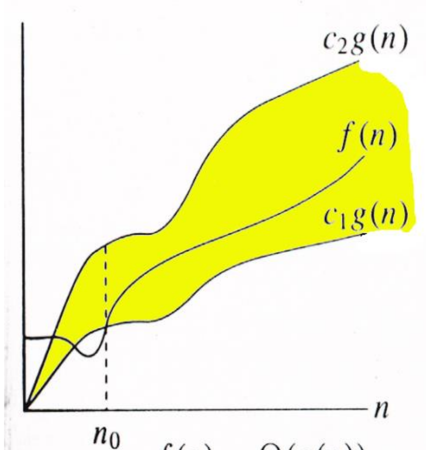
Short-cut:

- Drop low-order terms; ignore leading constants.
- Example: $3n^3 + 90n^2 - 5n + 6046 = \Theta(n^3)$

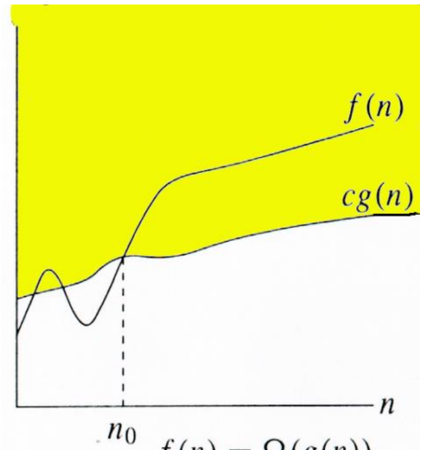
Relations Between Θ, *O*, Ω



$f(n) = O(g(n))$



$f(n) = \Theta(g(n))$



$f(n) = \Omega(g(n))$

For the functions f(n)=logn and g(n)=lgn. Which is true?

- $f(n)$ is $O(g(n))$
- $f(n)$ is $\Theta(g(n))$
- $f(n)$ is $\Omega(g(n))$
- All of the above

For the functions f(n)=logn and g(n)=lgn. Which is true?

- *f(n) is O(g(n))*
- *f(n) is Θ(g(n))*
- *f(n) is Ω(g(n))*
- **All of the above**

$$lgn = log_2 n = \frac{logn}{log2} = c_1\ logn$$

$$logn = log2(lgn) = c_2\ lgn$$

Benchmarking

- Algorithmic analysis is the first and best way, but not the final word
- What if two algorithms are both of the same complexity?
- Example: bubble sort and insertion sort are both $O(n^2)$
  - So, which one is the "faster" algorithm?
  - Benchmarking: run both algorithms on the same machine
  - Often indicates the constant multipliers and other "ignored" components
  - Still, different implementations of the same algorithm often exhibit different execution times – due to changes in the constant multiplier or other factors (such as adding an early exit to bubble sort)