Complexity

People naturally do comparisons

e.g. Car A is better than Car B!

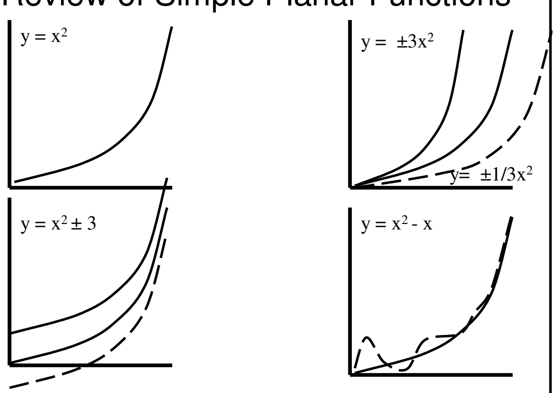
What is "better"?
  A has more horsepower than B.
  B holds 7 people, A only holds 2
Each can be better in a specific, useful context.

Can we do this with programs?

YES, using "Big **O**" Notation.

Label code to allow comparisons
What does ***better*** mean here?

Review of Simple Planar Functions



A review of simple, planar functions:

What decides the shape of the curve?

This is independent of:
  the lower order terms
  the coefficients used

Deriving work done by code

Given a specific language,
  a specific operating system and
  a specific compiler:

Consider this assignment:

x = x+1

How much time does it take?

Deriving work done by code

- Suppose we change the system?

- Suppose:

```
for ( int i=1; i<=n; i++ )
    x=x+1;
```

- Suppose:

```
for ( int i=1; i<=n; i++ )
    for ( int j=1; j<=n; j++)
        x=x+1;
```

Deriving work done by code

For any piece of code, generate a function to represent the work done:

For example:

$f(n) = c_1n + c_2 + c_3n + c_4n^3 + c_5n^2 + c_6 + c_7n^2 + c_8$

.

Simplifying:

$f(n) = c_4n^3 + (c_5 + c_7)n^2 + (c_1 + c_3)n + (c_2 + c_6 + c_8)$

Deriving work done by code

- **This is messy to graph.**

If all we are interested in is the basic shape, we can simplifyby using the dominant term.

This gives us a label to use for the code whose work is represented by f(x).

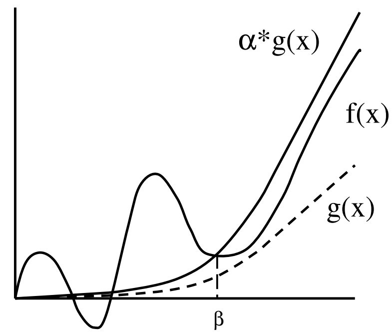
Definition: Upper Bound

- Given two functions f(n) and g(n) and two real constants α and β;

if α*g (n) ≥ f(n) , for all n>β
then g(n) is an <u>upper bound</u> for f(n)

f is said to be **O**(g(n))

Definition: Upper Bound



Upper Bounds

In particular, if f(n) is a polynomial then g(n) is the dominant term.

g(n) is an estimate of how f(n) acts.

We are guaranteed that f will do no worse than g. It might do better.

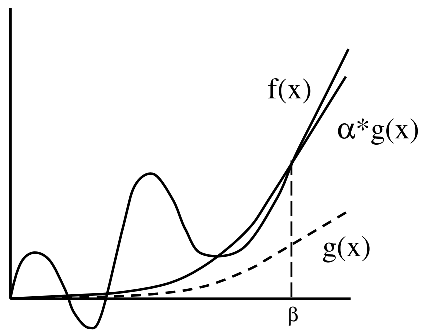
Definition: Lower Bound

- Given two functions f(n) and g(n) and two real constants α and β,

if $\alpha * g(n) \le f(n)$, for all $n > \beta$
then g(n) is a lower bound for f(n)

f is said to be $\Omega(g(n))$

Definition: Lower Bound



Lower Bounds

In particular, if f(n) is a polynomial then g(n) is the dominant term.

g(n) is an estimate of how f(n) acts.

We are guaranteed that f is no better than g. It might be worse.

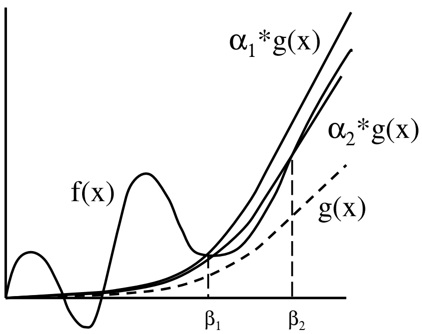
Both

- If f(n) is **O**(g(n)) and **Ω**(g(n)) then f is said to be **Θ**(g(n))

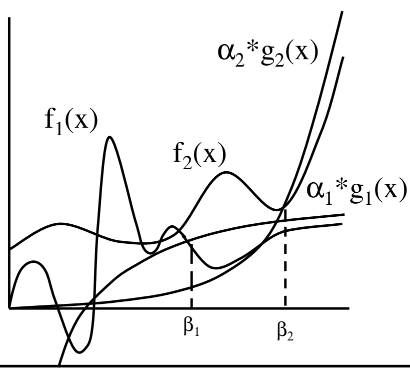
g is an upper bound for f <u>and</u> g is a lower bound for f.

e.g. $\alpha_2 * g(n) \le f(n) \le \alpha_1 * g(n)$

Definition: Both Bounds

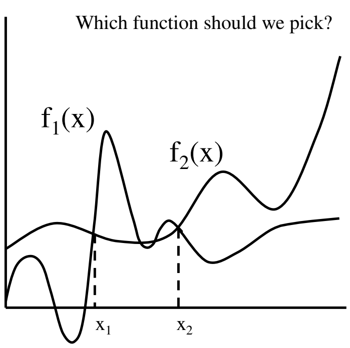


Example

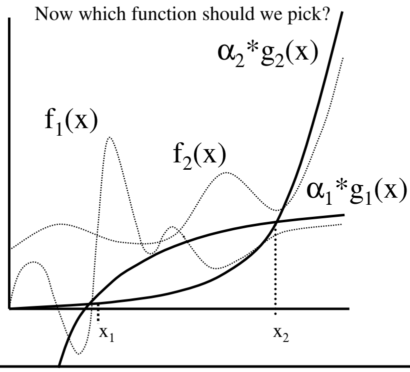


Example

Which function should we pick?



Example



Practical Issues

Most books use "Big **O**"
Sometimes "Big Theta"
Sometimes "Big Omega"
Usually we look at the time required.
Sometimes we look at the space required.
Remember, g is only a bound beyond the specified point **β**

More Practical Issues

Traditionally

- log n implies base 10
- lg n implies base 2
- ln n implies base e
- Other bases are specified $\log_b$ n

More Practical Issues

Sometimes, log n is used but $\log_2 n$ or lg n is implied by context.

- The identity $\log_b a = \dfrac{\log_c a}{\log_c b}$

makes the conversion a constant

Standard "Big O" Values

<u>Polynomial time</u> (P)

<u>Nondeterministically Polynomial Time</u> (NP)

Computer scientists believe $P \subseteq NP$.
This is not proven.

<u>"Big O": Polynomial time</u> (P)

O(1)	constant time
O(log n)	log time
O(n)	linear time
O(nlogn)	
O(n^2)	quadratic time
O(n^3)	cubic time
:	
O(n^k)	

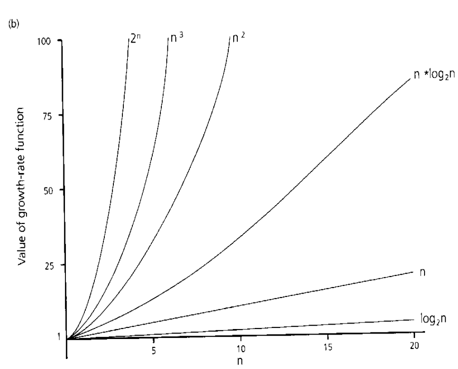
“Big O”: NP Time

Nondeterministically Polynomial Time

O(n!) factorial time
O(2^n) exponential time

If P ⊆ NP, then, it means that some problems can **never** be solved quickly.

Function	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n \log_2 n$	30	364	9965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$



- Tune in to 605.421 for more exciting complexity theory.
- In this course, algorithms range from constant time to cubic time,

  e.g. Sorts range from $O(n)$ to $\mathbf{O}(n^2)$

Remember:

$O(g(n))$ is an <u>estimate</u> of performance.

Possibly:

The "worst" algorithm is sometimes the best choice