Chap3. Growth of Functions

- Asymptotic notation
- Comparison of functions
- Standard notations and common functions

Asymptotic notation

- How do you answer the question: “what is the running time of algorithm *x*?”
- We need a way to talk about the computational cost of an algorithm that focuses on the essential parts and ignores irrelevant details
- We’ve seen some of this already:
  - *$O(n^2)$ for insertion sort*
  - *$\Theta(n\log n)$ for merge sort*

Asymptotic notation

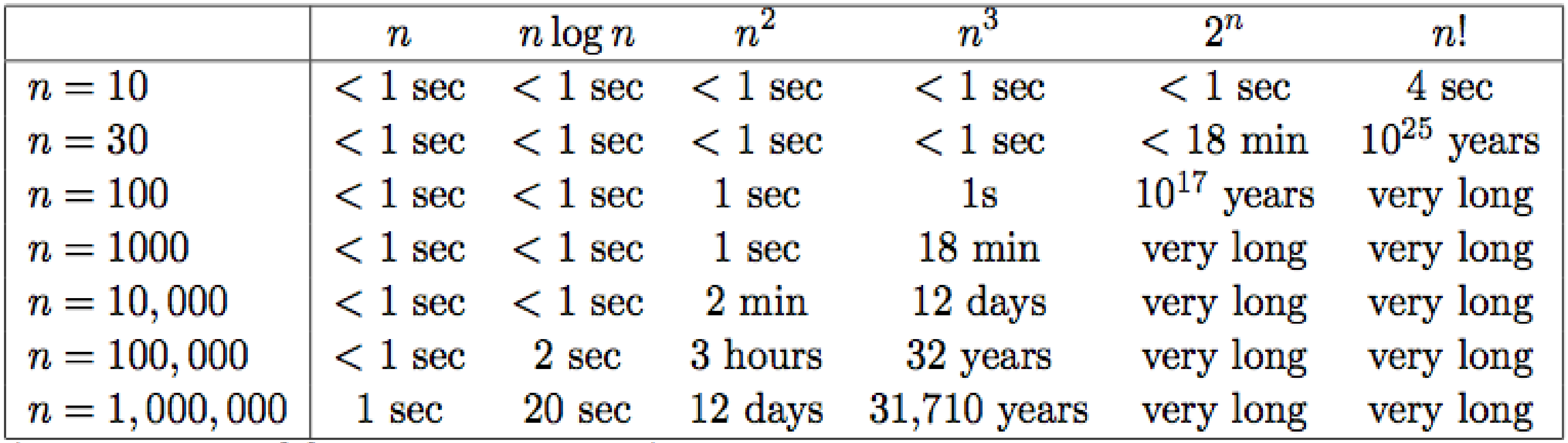
- Precisely calculating the actual steps is tedious and not generally useful
- Different operations take different amounts of time. Even from run to run, things such as caching, etc. cause variations
- Want to identify categories of algorithmic runtimes

For example…

- $f_1(n)$ takes n^2 steps
- $f_2(n)$ takes $2n + 100$ steps
- $f_3(n)$ takes $3n+1$ steps

- Which algorithm is better?
- Is the difference between f_2 and f_3 important/significant?

Runtime examples



	n	$n \log n$	n^2	n^3	2^n	$n!$
$n = 10$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	4 sec
$n = 30$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 18 min	10^{25} years
$n = 100$	$<$ 1 sec	$<$ 1 sec	1 sec	1s	10^{17} years	very long
$n = 1000$	$<$ 1 sec	$<$ 1 sec	1 sec	18 min	very long	very long
$n = 10,000$	$<$ 1 sec	$<$ 1 sec	2 min	12 days	very long	very long
$n = 100,000$	$<$ 1 sec	2 sec	3 hours	32 years	very long	very long
$n = 1,000,000$	1 sec	20 sec	12 days	31,710 years	very long	very long

(adapted from [2], Table 2.1, pg. 34)

Asymptotic notation

- What does asymptotic mean?
  Asymptotic describes the behavior of a function in the limit (for sufficiently large values of its parameter)

Asymptotic notation

- The order of growth of the running time of an algorithm is defined as the highest-order term (usually the leading term) of an expression that describes the running time of the algorithm.
- We ignore the highest-order term's coefficient, as well as all of the lower order terms in the expression.
- Example: The order of growth of an algorithm whose running time is described by the expression $an^2 + bn + c$ is simply n^2.

Big O: Upper bound

- *O(g(n))* is the set of functions:

$$O(g(n)) = \left\{ f(n): \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

Big O: Upper bound

- *O(g(n))* is the set of functions:

$$O(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

We can bound the function *f(n)* above by some constant factor of *g(n)*

Big O: Upper bound

- *O(g(n))* is the set of functions:

$$O(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

We can bound the function *f(n)* above by some constant multiplied by *g(n)*

For some increasing range

Big O: Upper bound

- *O(g(n))* is the set of functions:

$$O(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

<u>Generally, we’re most interested in big O notation since it is an upper bound on the running time</u>

Big O: examples

- $7n-2$ is $O(n)$
  need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq c{\cdot}n$ for $n \geq n_0$

  this is true for $c = 7$ and $n_0 = 1$
- $3n^3 + 20n^2 + 5$ is $O(n^3)$
  need $c > 0$ and $n_0 \geq 1$ s.t $3n^3 + 20n^2 + 5 \leq c{\cdot}n^3$ for $n \geq n_0$
  this is true for $c = 4$ and $n_0 = 21$
- $3 \log n + 5$ is $O(\log n)$
  need $c > 0$ and $n_0 \geq 1$ s.t $3 \log n + 5 \leq c{\cdot}\log n$ for $n \geq n_0$
  this is true for $c = 8$ and $n_0 = 2$

Omega: Lower bound

- *Ω(g(n))* is the set of functions:

$$\mathrm{W}(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

Omega: Lower bound

- *Ω(g(n))* is the set of functions:

$$W(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

We can bound the function *f(n)* below by some constant factor of *g(n)*

Theta: Upper and lower bound

- *Θ(g(n))* is the set of functions:

$$\Theta(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c_1, c_2 \text{ and } n_0 \text{ such that} \\ 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

Theta: Upper and lower bound

- *Θ(g(n))* is the set of functions:

$$\Theta(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c_1, c_2 \text{ and } n_0 \text{ such that} \\ 0 \le \boxed{c_1 g(n) \le f(n) \le c_2 g(n)} \text{ for all } n \ge n_0 \end{array} \right\}$$

We can bound the function *f(n)* above and below by some constant factor of *g(n)* (though different constants)

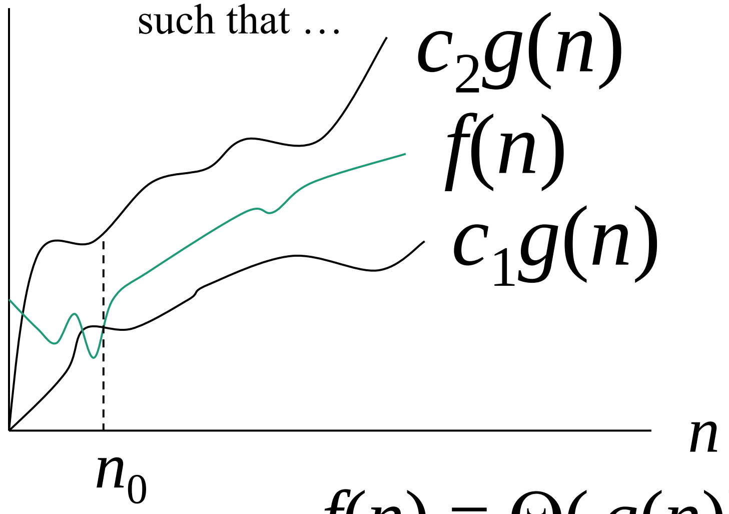
Theta: Upper and lower bound

- *Θ(g(n))* is the set of functions:

$$\Theta(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c_1, c_2 \text{ and } n_0 \text{ such that} \\ 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

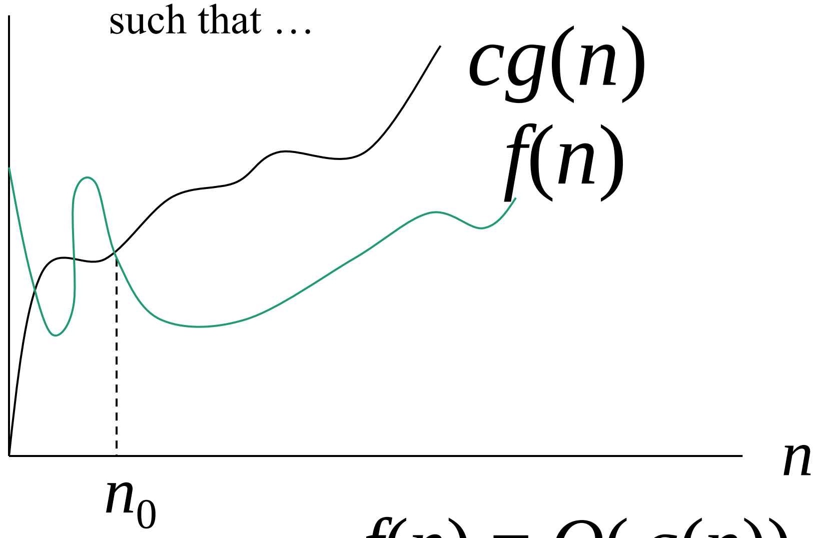
Note: A function is theta bounded *iff* it is big O bounded and Omega bounded

There exist positive constants c_1 and c_2 such that there is a positive constant n_0 such that …



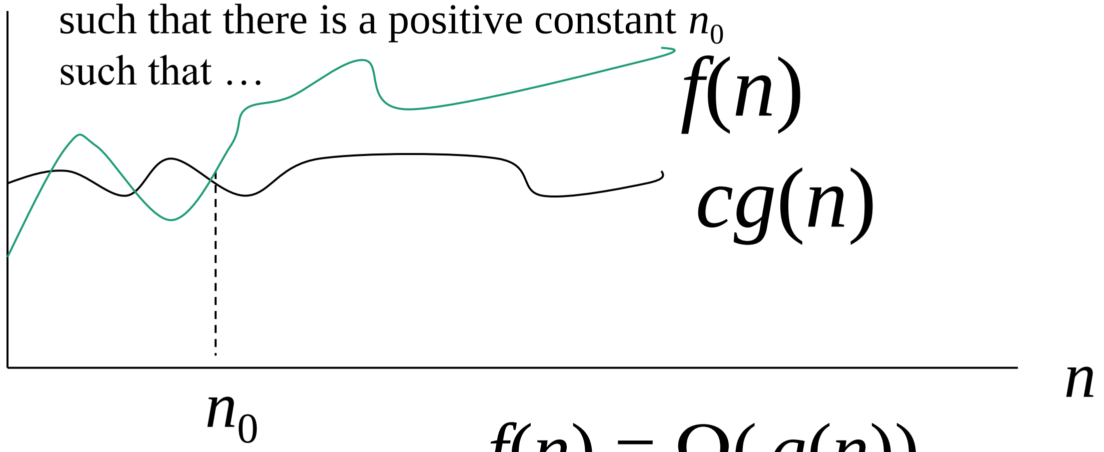
$$f(n) = \Theta(g(n))$$

There exist positive constants c such that there is a positive constant n_0 such that …



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

There exist positive constants c such that there is a positive constant n_0 such that …



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

Some rules of thumb

- Multiplicative constants can be omitted
  - $14n^2$ becomes n^2
  - 7 log n becomes log n
- Lower order functions can be omitted
  - $n + 5$ becomes n
  - $n^2 + n$ becomes n^2
- n^a dominates n^b if $a > b$
  - n^2 dominates n, so n^2+n becomes n^2
  - $n^{1.5}$ dominates $n^{1.4}$, so $n^{1.5}+n^{1.4}$ *becomes* $n^{1.5}$

Some rules of thumb

- a^n dominates b^n if $a > b$
  - 3^n dominates 2^n
- Any exponential dominates any polynomial
  - 3^n dominates n^5
  - 2^n dominates n^c
- Any polynomial dominates any logarithm
  - n dominates log n or log log n
  - n^2 dominates n log n
  - $n^{1/2}$ dominates *log n*
- <u>Do not omit lower order terms of different variables ($n^2 + m$) does not become n^2</u>

Some examples

- O(1) : constant. Fixed amount of work, regardless of the input size
    - add two 32 bit numbers
    - determine if a number is even or odd
    - sum the first 20 elements of an array
    - delete an element from a doubly linked list
- O(log *n*) : logarithmic. At each iteration, discards some portion of the input (i.e. half)
    - binary search

Some examples

- O(*n*): linear. Do a constant amount of work on each element of the input
  - find an item in a linked list
  - determine the largest element in an array

- O(*n* log *n*):log-linear. Divide and conquer algorithms with a linear amount of work to recombine
  - Sort a list of number with Merge Sort

Some examples

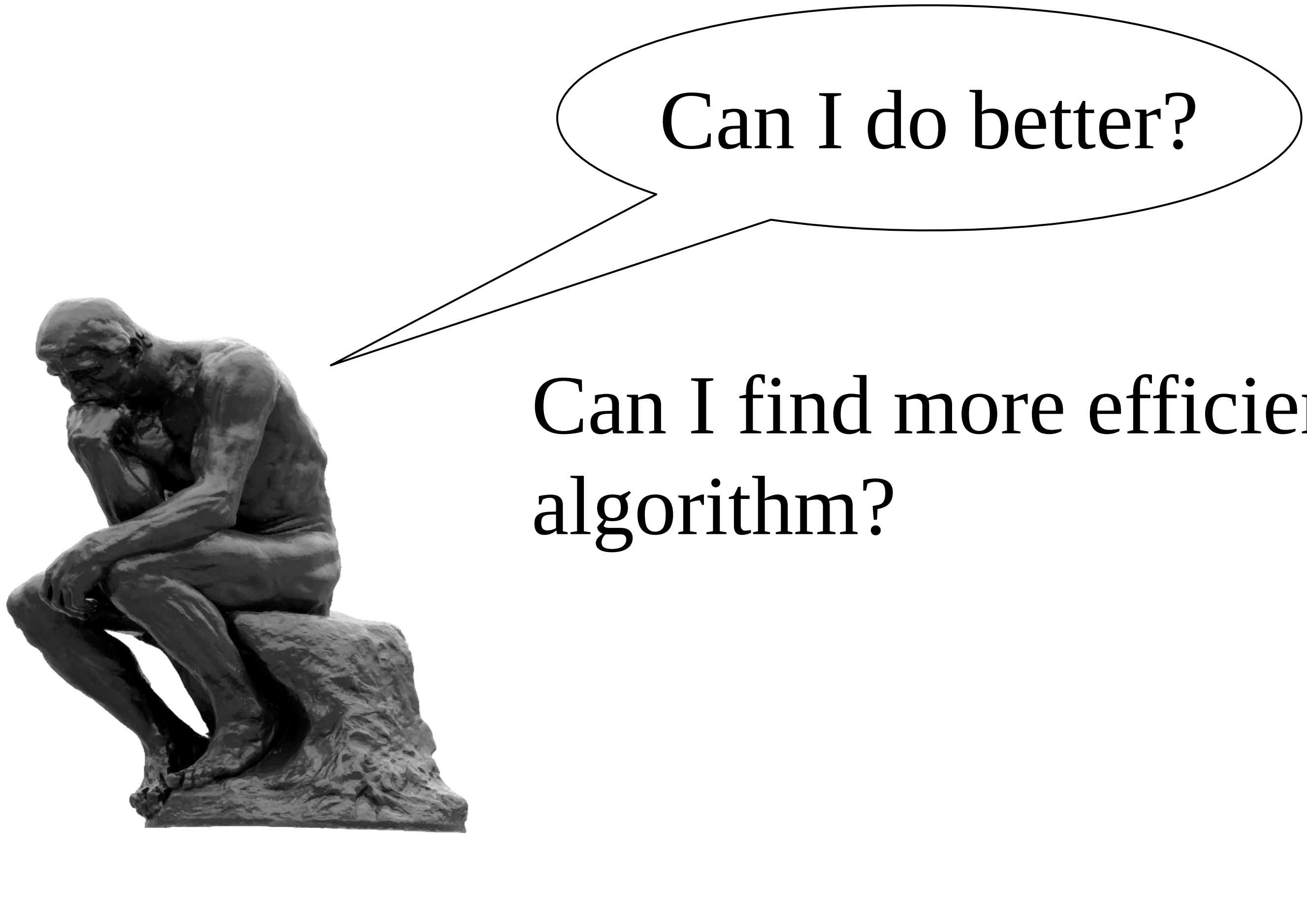
- O(n^2) : quadratic. Double nested loops that iterate over the data
  - Insertion sort, selection sort, …..
- O(n^3) : cubic. Triple nested loops that iterate over the data

- O(2^n) : exponential
  - Enumerate all possible subsets
  - Traveling salesman using dynamic programming
- O(n!)
  - Enumerate all permutations

Big O, Omega, Theta

- Why would we prefer to express the running time of merge sort as $\Theta(n(\log_2 n))$ instead of $O(n(\log_2 n))$?
    - Because Theta is ***more precise*** than Big O.
- If we say that the running time of merge sort is $O(n(\log_2 n))$, we are merely making a claim about merge sort's asymptotic upper bound, whereas of we say that the running time of merge sort is $\Theta(n(\log_2 n))$, we are making a claim about merge sort's asymptotic *upper and lower bounds.*

Algorithm Designer



Can I find more efficient algorithm?

Summary

- Asymptotic notation
- Big O, Omega, Theta
- Some rules of thumb
- O(*1*), O(*logn*), O(*n*), O(*nlogn*), O(n^2), O(2^n), O(n!)