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 ru nning 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:
  - linear
  - $n \log n$
  - n^2

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)

Big O: Upper bound

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

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

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 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 increasi ng range

Big O: Upper bound

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

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

$$O(n^2) = \begin{array}{lcl} f_1(x) & = & 3n^2 \\ f_2(x) & = & 1/2n^2 + 100 \\ f_3(x) & = & n^2 + 5n + 40 \\ f_4(x) & = & 6n \end{array}$$

Big O: Upper bound

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

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

Generally, we're most interested in big O notation since it is an upper bound on the running time

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:

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

Omega: Lower bound

- *Ω(g(n))* is the set of functions:

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

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

Omega: Lower bound

- $\Omega(g(n))$ is the set of functions:

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

$$\Omega(n^2) = \begin{array}{lcl} f_1(x) & = & 3n^2 \\ f_2(x) & = & 1/2n^2 + 100 \\ f_3(x) & = & n^2 + 5n + 40 \\ f_4(x) & = & 6n^3 \end{array}$$

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:

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

We can bound the function *f(n)* above and below by some const ant factor of *g(n)* (though differe nt 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

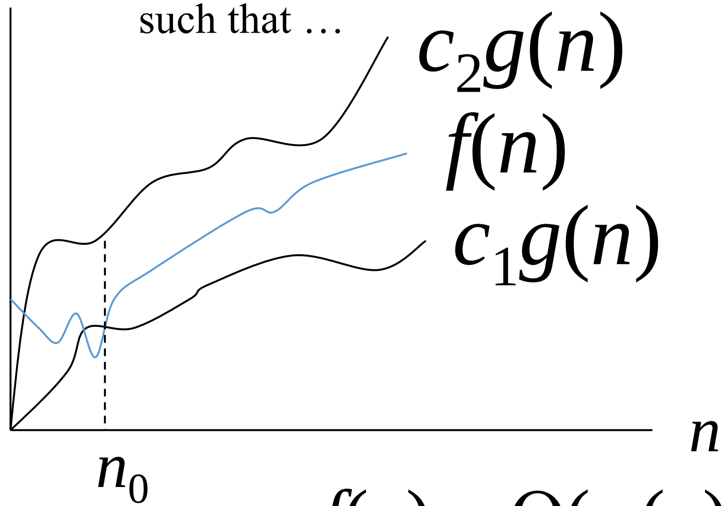
Theta: Upper and lower bound

- *Θ(g(n))* is the set of functions:

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

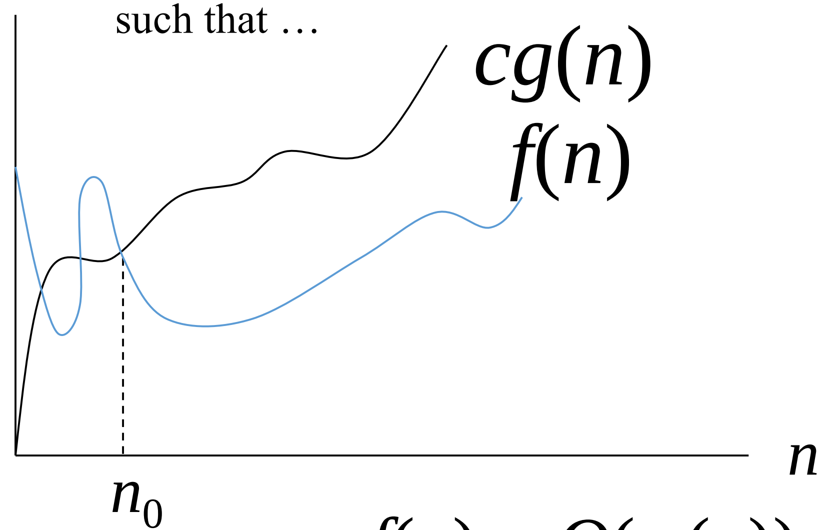
$$\Theta(n^2) = \begin{array}{lcl} f_1(x) & = & 3n^2 \\ f_2(x) & = & 1/2n^2 + 100 \\ f_3(x) & = & n^2 + 5n + 40 \\ f_4(x) & = & 3n^2 + n \log n \end{array}$$

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



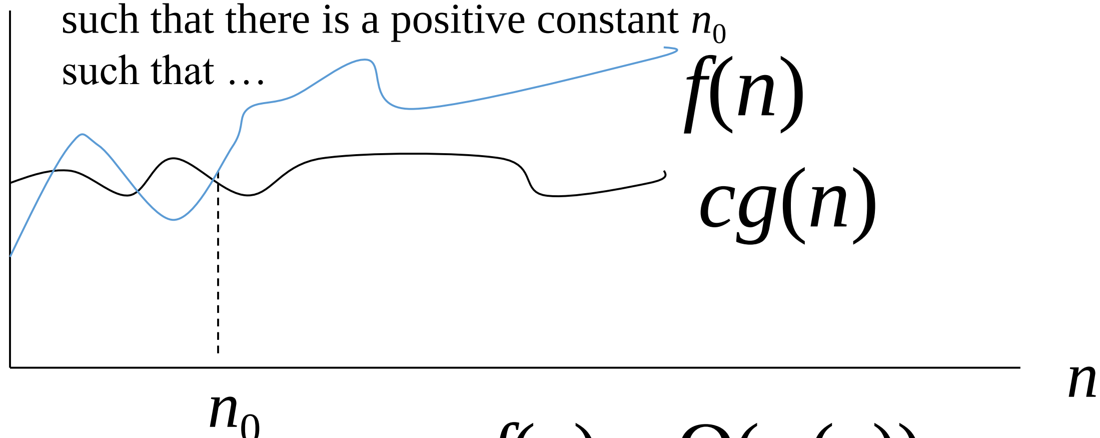
$f(n) = \Theta(g(n))$

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 …



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

$\Theta()$ proofs

- Prove $\boxed{\frac{1}{2}n^2 - 3n = \Theta(n^2)}$
  - Find c_1, c_2 and n_0 such that

$$c_1 n^2 \le \frac{1}{2}n^2 - 3n \le c_2 n^2$$

$$c_1 \le \frac{1}{2} - \frac{3}{n} \le c_2$$

$$\frac{1}{2} - \frac{3}{n} \le c_2 \rightarrow n \ge 1, c_2 \ge \frac{1}{2}$$

$$c_1 \le \frac{1}{2} - \frac{3}{n} \rightarrow n \ge 7, c_1 \le \frac{1}{14}$$

$$c_1 = \frac{1}{14}$$

$$c_2 = \frac{1}{2}$$

$$n_0 = 7$$

Proving bounds (find constants that satisfy inequalities)

- Show that $5n^2 - 15n + 100$ is $\Theta(n^2)$
- Step 1: Prove $O(n^2)$
  - Find constants c and n_0 such that $5n^2 - 15n + 100 \leq cn^2$ for all n > n_0

$$cn^2 \geq 5n^2 - 15n + 100$$

$$c \geq 5 - 15/n + 100/n^2$$

Let n_0 =1 and c = 5 - 15 +100 = 90.
100/n^2 only get smaller as n increases and we ign ore -15/n since it only varies between -15 and 0

Proving bounds

- Step 2: Prove $\Omega(n^2)$
  - Find constants c and n_0 such that $5n^2 - 15n + 100 \geq cn^2$ for all n > n_0

$$cn^2 \leq 5n^2 - 15n + 100$$

$$c \leq 5 - 15/n + 100/n^2$$

Let n_0 =4 and c = 5 – 15/4 = 1.25 (or anything less than 1.25). We can ignore $100/n^2$ since it is always positive and 15/n is always decreasing.

Bounds

Is $5n^2 \ O(n)$?   No!

How would we prove it?

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

Disproving bounds

Is $5n^2 \;\; O(n)$?

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

Assume it's true. That means there exists some c and n_0 such that

$$5n^2 \le cn \text{ for } n > n_0$$

$$5n \le c$$ contradiction!

Some rules of thumb

- Multiplicative constants can be omitted
  - $14n^2$ becomes n^2
  - 7 log n become 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 $n^{1.4}$

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</u> ($n^2 + m$) does not become n^2

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
  - FFT

Some examples

- $O(n^2)$: quadratic. Double nested loops that iterate over the data
  - Insertion sort, selection sort, ...
- $O(2^n)$: exponential
  - Enumerate all possible subsets
  - Traveling salesman using dynamic programming
- O(n!)
  - Enumerate all permutations

Asymptotic Notation

- What does asymptotic mean?
  Asymptotic describes the behavior of a function *in the limit* (for sufficiently large values of its parameter).
- 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.
- Example: The order of growth of an algorithm whose running time is described by the expression $an^2 + bn + c$ is simply n^2.

- Let's say that we have some function that represents the sum total of all the running-time costs of an algorithm; call it *f(n)*.
- For merge sort, the actual running time is: $f(n) = cn(\log_2 n) + cn$
- We want to describe the running time of merge sort in terms of another function, *g(n), so that* we can say *f(n) = O(g(n)), like this:* $cn(\log_2 n) + cn = O(n\log_2 n)$

Big O, Omega, Theta

- Already studied
- 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 Big-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.*