Analysis of Algorithms

- Measures of efficiency:
  - Running time
  - Space used
  - others
- Efficiency as a function of input size (NOT value!)
  - Number of data elements (numbers, points)
  - Number of bits in an input number

    e.g. Find the factors of a number n,

    Determine if an integer n is prime

Model: What machine do we assume? Intel? Motorola? P3? P4?

The RAM model

- Generic abstraction of sequential computers
- RAM assumptions:
  - Instructions (each taking constant time), we usually choose one type of instruction as a **characteristic** operation that is counted:
    - Arithmetic (add, subtract, multiply, etc.)
    - Data movement (assign)
    - Control (branch, subroutine call, return)
    - Comparison
  - Data types – integers, characters, and floats
  - Ignores memory hierarchy, network!

Importance of input size

Consider the problem of factoring an integer n

Note: Public key cryptosystems depend critically on hardness of factoring – if you have a fast algorithm to factor integers, most e-commerce sites will become insecure!!

Trivial algorithm: Divide by 1,2,…, n/2 (n/2 divisions)

aside: think of an improved algorithm

Always evaluate running time as a function of the SIZE of the input (e.g. in the number of bits or the number of ints, or number of floats, or number of chars,…)

Analysis of Find-max

- COUNT the number of cycles (**running time**) as a function of the **input size**

Find-max (A)	cost	times
1. max ← A[1]	c_1	1
2. for j ← 2 to length(A)	c_2	n
3. do if (max < A[j])	c_3	n-1
4. max ← A[j]	c_4	$0 \le k \le n-1$
5. return max	c_5	1

Running time (upper bound): $c_1 + c_5 - c_3 - c_4 + (c_2 + c_3 + c_4)n$

Running time (lower bound): $c_1 + c_5 - c_3 - c_4 + (c_2 + c_3)n$

Q: What are the values of c_i?

Best/Worst/Average Case Analysis

- **Best case**: A[1] is the largest element.
- **Worst case**: elements are sorted in increasing order
- **Average case**: ? Depends on the input characteristics

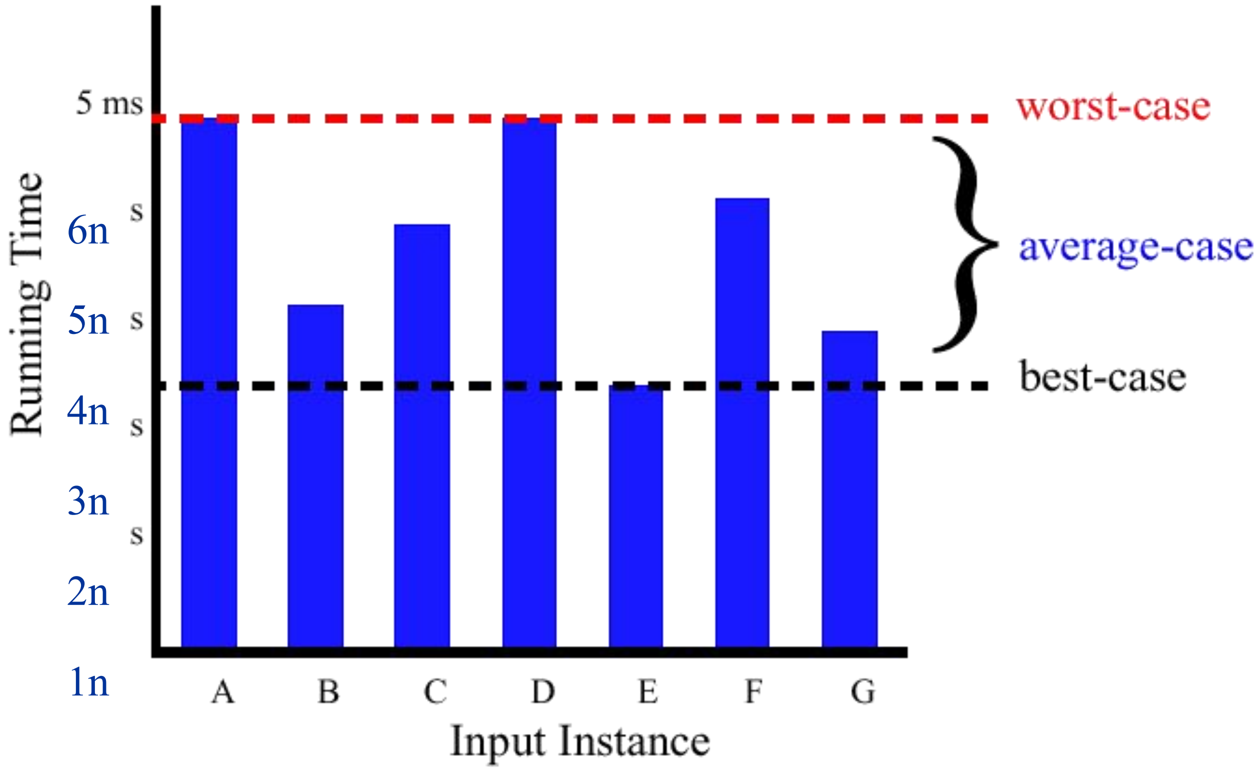
Q: What do we use?

A: Worst case or Average-case is usually used:

- Worst-case is an upper-bound; in certain application domains (e.g., air traffic control, surgery) knowing the **worst-case** time complexity is of crucial importance
- Finding the **average case** can be very difficult; needs knowledge of input distribution.
- Best-case is not very useful.

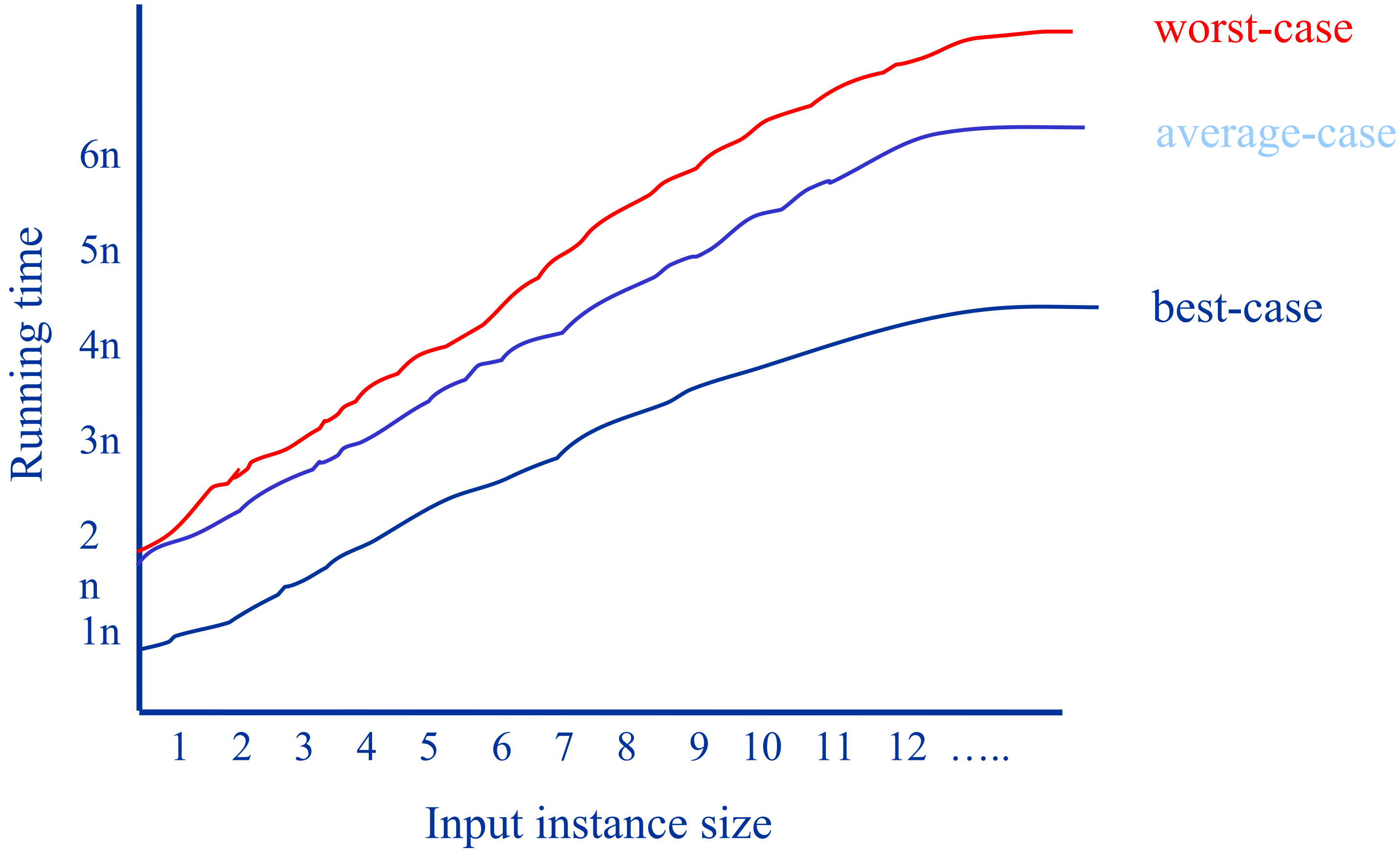
Best/Worst/Average Case (2)

- For a specific size of input *n*, investigate running times for different input instances:



Best/Worst/Average Case (3)

– For inputs of all sizes:



Asymptotic notation : Intuition

Running time bound: $c_1 + c_5 - c_3 - c_4 + (c_2 + c_3 + c_4)n$
What are the values of c_i? machine-dependent

A simpler expression: $c_5 + c_6 n$ [**still complex**].

Q: Can we throw away the lower order terms?
A: Yes, if we do not worry about constants, and there exist constants c_7, c_8 such that $c_7 n \le c_5 + c_6 n \le c_8 n$, then we say that the running time is $\theta(n)$.

Need some mathematics to formalize this (LATER).

Q: Are we interested in small n or large?
A: Assume interested in large n – cleaner theory, usually realistic. Remember the assumption when interpreting results!

Asymptotic notation - continued

Will do the relevant math later. For now, the intuition is:

1. O() is used for upper bounds "grows slower than"
2. Ω() used for lower bounds "grows faster than"
3. Θ() used for denoting matching upper and lower bounds. "grows as fast as"

These are bounds on running time, not for the problem

The thumbrules for getting the running time are

1. Throw away all terms other than the most significant one -- Calculus may be needed

   e.g.: which is greater: n log n or $n^{1.001}$?

2. Throw away the constant factor.
3. The expression is Θ() of whatever's left.

   Asymptotic optimality – expression inside Θ() best possible.

A Harder Problem

INPUT: A[1..n] - an array of integers, k, $1 \leq k \leq length(A)$

OUTPUT: an element m of A such that m is the k^{th} largest element in A.

Think for a minute

Brute Force: Find the maximum, remove it. Repeat k-1 times. Find maximum.

Q: How good is this algorithm?

A: Depends on k! Can show that the running time is $\Theta(nk)$. If k=1, asymptotically optimal.

Also true for any constant k.

If k = log n, running time is $\Theta(n \log n)$. Is this good?

If k = n/2 (MEDIAN), running time is $\Theta(n^2)$.

Definitely bad! Can sort in $O(n \log n)$!

Q: Is there a better algorithm? YES!

Analysis of Insertion Sort

Let's compute the **running time** as a function of the **input size**

	cost	**times**
for j←2 **to** *n*	c_1	n
do key←A[j]	c_2	n-1
Insert A[j] into the sorted sequence A[1..j-1]	0	n-1
i←j-1	c_3	n-1
while i>0 **and** A[i]>key	c_4	$\sum_{j=2}^{n} t_j$
do A[i+1]←A[i]	c_5	$\sum_{j=2}^{n} (t_j - 1)$
i ← i-1	c_6	$\sum_{j=2}^{n} (t_j - 1)$
A[i+1] ← key	c_7	n-1

Analysis of Insertion Sort - contd

- **Best case**: elements already sorted $\rightarrow t_j=1$, running time = $f(n)$, i.e., *linear* time.
- **Worst case**: elements are sorted in inverse order $\rightarrow t_j=j$, running time = $f(n^2)$, i.e., *quadratic* time
- **Average case**: $t_j=j/2$, running time = $f(n^2)$, i.e., *quadratic* time

- We analyzed insertion sort, and it has worst case running time $An^2 + Bn + C$, where $A = (c_5+c_6+c_7)/2$ etc.
- Q1: How useful are the details in this result?
- Q2: How can we simplify the expression?

Back to asymptotics…

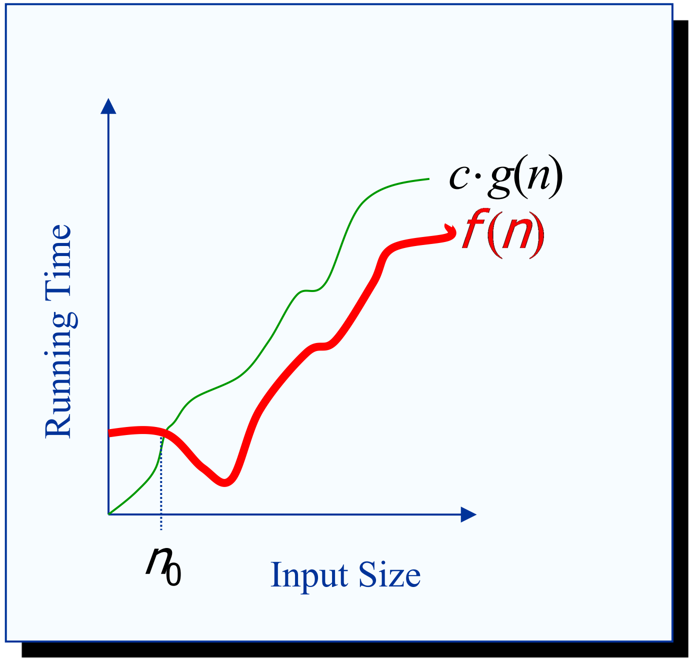
We will now look more formally at the process of simplifying running times and other measures of complexity.

Asymptotic analysis - details

- Goal: to simplify analysis of running time by getting rid of ”details”, which may be affected by specific implementation and hardware
  - like “rounding”: 1,000,001 » 1,000,000
  - $3n^2$ » n^2
- Capturing the essence: how the running time of an algorithm increases with the size of the input *in the limit.*
  - Asymptotically more efficient algorithms are best for all but small inputs

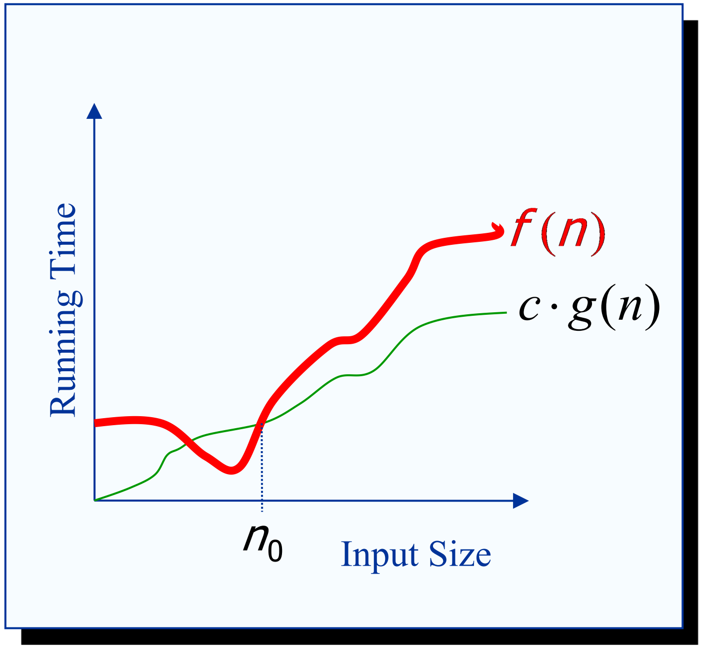
Asymptotic notation

- The "big-Oh" *O*-Notation
  - asymptotic upper bound
  - $f(n) \in O(g(n))$, if there exists constants *c* and n_0, *s.t.* $\mathbf{f(n) \le c\ g(n)}$ for $n \ge n_0$
  - $f(n)$ and $g(n)$ are functions over non-negative integers
- Used for *worst-case* analysis



Asymptotic notation - contd

- The "big-Omega" Ω–Notation
  - asymptotic lower bound
  - $f(n) \in \Omega(g(n))$ if there exists constants c and n_0, *s.t.* $\mathbf{c\ g(n) \leq f(n)}$ for $n \geq n_0$
- Used to describe *best-case* running times or lower bounds of algorithmic problems
  - E.g., lower-bound of searching in an unsorted array is $\Omega(n)$.



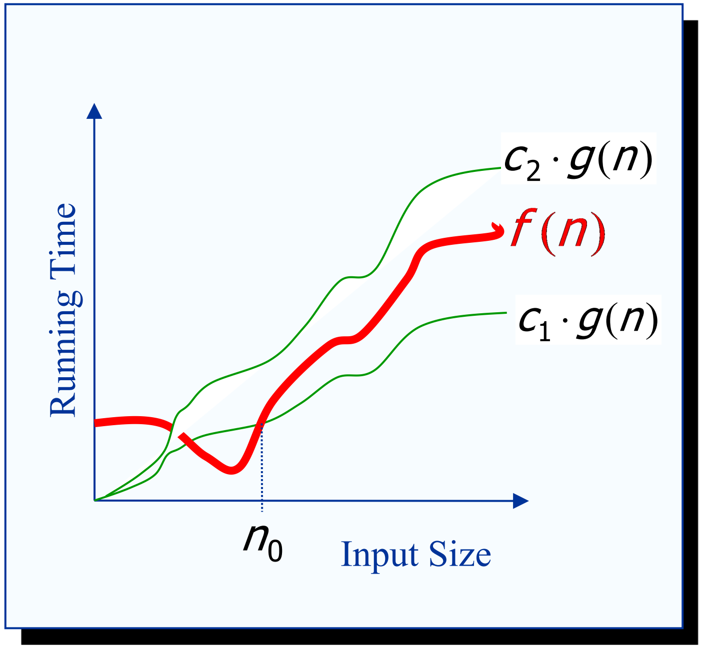
Asymptotic notation - contd

- Simple Rule: Drop lower order terms and constant factors.
  - $50\, n \log n \in O(n \log n)$
  - $7n - 3 \in O(n)$
  - $8n^2 \log n + 5n^2 + n \in O(n^2 \log n)$

- Note: Even though $50\, n \log n \in O(n^5)$, we usually try to express a O() expression using as small an order as possible

Asymptotic notation - contd

- The "big-Theta" Θ–Notation
  - asymptoticly tight bound
  - *f(n)* $\in \Theta(g(n))$ if there exists constants c_1, c_2, and n_0, *s.t.* $c_1\, g(n) \le f(n) \le c_2\, g(n)$ for $n \ge n_0$
- *f(n)* $\in \Theta(g(n))$ if and only if *f(n)* $\in O(g(n))$ and *f(n)* $\in \Omega(g(n))$
- *O(f(n))* is often misused instead of $\Theta(f(n))$



Asymptotic Notation - contd

- Two more asymptotic notations
  - 'Little-Oh' notation $f(n)=o(g(n))$
    non-tight analogue of Big-Oh
    - For every c, there should exist n_0, s.t. $\mathbf{f(n) \le c\, g(n)}$ for $n \ge n_0$
    - Used for **comparisons** of running times.
      If $f(n) \in o(g(n))$, it is said that *g(n) dominates f(n).*
    - More useful defn:
      (uses calculus)

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

  - 'Little-omega' notation $f(n) \in \omega(g(n))$
    non-tight analogue of Big-Omega

Asymptotic Notation - contd

- (VERY CRUDE) Analogy with real numbers
  - $f(n) = O(g(n))$ ≅ $f \leq g$
  - $f(n) = \Omega(g(n))$ ≅ $f \geq g$
  - $f(n) = \Theta(g(n))$ ≅ $f = g$
  - $f(n) = o(g(n))$ ≅ $f < g$
  - $f(n) = \omega(g(n))$ ≅ $f > g$
- Abuse of notation: $f(n) = O(g(n))$ actually means $f(n)$ Î $O(g(n))$.

Asymptotic Notation - contd

Common uses:

$\Theta(1)$ – *constant.*

$n^{\Theta(1)}$ – *polynomial*

$2^{\Theta(n)}$ – *exponential*

Be careful!

$n^{\Theta(1)} \neq \Theta(n^1)$

$2^{\Theta(n)} \neq \Theta(2^n)$

- When is asymptotic analysis useful?
- When is it NOT useful?

Many, many abuses of asymptotic notation in Computer Science literature.

Lesson: Always remember the implicit assumptions…

Comparison of running times

Running Time	Maximum problem size (n)		
	1 second	1 minute	1 hour
$400n$	2500	150000	9000000
$20n \log n$	4096	166666	7826087
$2n^2$	707	5477	42426
n^4	31	88	244
2^n	19	25	31

Classifying functions

$T(n)$	10	100	1,000	10,000
$\log n$	3	6	9	13
$n^{1/2}$	3	10	31	100
n	10	100	1,000	10,000
$n \log n$	30	600	9,000	130,000
n^2	100	10,000	10^6	10^8
n^3	1,000	10^6	10^9	10^{12}
2^n	1,024	10^{30}	10^{300}	10^{3000}

Logarithmic functions

- $\log_{10} n$ = # digits to write n
- $\log_2 n$ = # bits to write n
  $= 3.32 \log_{10} n$
- $\log(n^{1000}) = 1000 \log(n)$

Differ only by a multiplicative constant.

Poly Logarithmic (a.k.a. polylog)

$$(\log n)^5 = \log^5 n$$

Crucial asymptotic facts

Logarithmic << Polynomial

$\log^{1000} n \ll n^{0.001}$ For sufficiently large n

Linear << Quadratic

$10000\, n \ll 0.0001\, n^2$ For sufficiently large n

Polynomial << Exponential

$n^{1000} \ll 2^{0.001\, n}$ For sufficiently large n

Are constant functions constant?

The running time of the algorithm is a "Constant": It does not depend **significantly** on the size of the input.

Yes • 5

Yes • 1,000,000,000,000

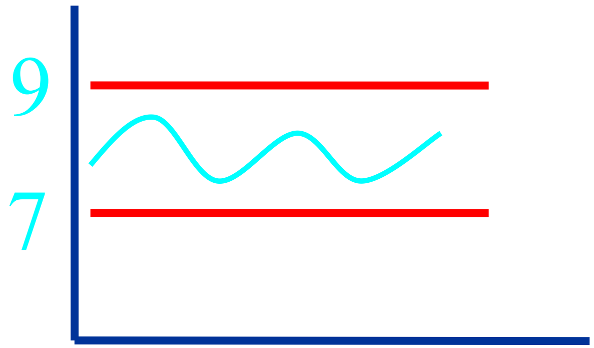
Yes • 0.000000000001

No • -5

No • 0

Yes • 8 + sin(n)

Write θ(1).



Lie in between

Polynomial functions

Quadratic

- n^2
- $0.001\ n^2$
- $1000\ n^2$
- $5n^2 + 3000n + 2\log n$

Lie in between

Polynomial

- n^c
- $n^{0.0001}$
- n^{10000}
- $5n^2 + 8n + 2\log n$
- $5n^2 \log n$
- $5n^{2.5}$

Lie in between

Exponential functions

- 2^n
- $2^{0.0001\,n}$
- $2^{10000\,n}$
- $8^n \quad = 2^{3n}$
- $2^n / n^{100} \quad > 2^{0.5n}$
- $2^n \cdot n^{100} \quad < 2^{2n}$

$$2^{0.5n} > n^{100}$$

$$2^n = 2^{0.5n} \cdot 2^{0.5n} > n^{100} \cdot 2^{0.5n}$$

$$2^n / n^{100} > 2^{0.5n}$$

Proving asymptotic expressions

Use definitions!

e.g. $f(n) = 3n^2 + 7n + 8 = \theta(n^2)$

$f(n) \in \Theta(g(n))$ if there exists constants c_1, c_2, and n_0, *s.t.*

$\mathbf{c_1\, g(n) \le f(n) \le c_2\, g(n)}$ for $n \ge n_0$

Here $g(n) = n^2$

One direction $(f(n) = \Omega(g(n))$ is easy

$c_1\, g(n) \le f(n)$ holds for $c_1 = 3$ and $n \ge 0$

The other direction $(f(n) = O(g(n))$ needs more care

$f(n) \le c_2\, g(n)$ holds for $c_2 = 18$ and $n \ge 1$ (CHECK!)

So $n_0 = 1$

Proving asymptotic expressions - 2

Caveats!

1. constants c_1, c_2 MUST BE POSITIVE.
2. Could have chosen $c_2 = 3 + \varepsilon$ for any $\varepsilon > 0$. WHY?

-- because $7n + 8 \leq \varepsilon n^2$ for $n \geq n_0$ for some sufficiently large n_0. Usually, the smaller the ε you choose, the harder it is to find n_0. So choosing a large ε is easier.

3. Order of quantifiers

$\exists c_1\ c_2\ \exists n_0\ \forall\ n \geq n_0,\ c_1 g(n) \leq f(n) \leq c_2 g(n)$

vs

$\exists n_0\ \forall\ n \geq n_0\ \exists c_1\ c_2,\ c_1 g(n) \leq f(n) \leq c_2 g(n)$

-- allows a different c_1 and c_2 for each n. Can choose $c_2 = 1/n$!! So we can "prove" $n^3 = \Theta(n^2)$.

Why polynomial vs exponential ?

Philosophical/Mathematical reason – polynomials have different properties, grow much slower; mathematically natural distinction.

Practical reasons

1. almost every algorithm ever designed and every algorithm considered practical are very low degree polynomials with reasonable constants.
2. a large class of natural, practical problems seem to allow only exponential time algorithms. Most experts believe that there do not exist any polynomial time algorithms for any of these.

Important thumbrules for sums

"addition made easy" - Jeff Edmonds.

"Theta of last term"

Geometric like: $f(i) = 2^{\Omega(i)} \Rightarrow \sum_{i=1}^{n} f(i) = \Theta(f(n))$

no of terms x last term

Arithmetic like: $i.f(i) = i^{\Theta(1)} \Rightarrow \sum_{i=1}^{n} f(i) = \Theta(nf(n))$

Harmonic: $f(i) = 1/i \Rightarrow \sum_{i=1}^{n} f(i) = \Theta(\log n)$

"Theta of first term"

Bounded tail: $i.f(i) = 1/i^{\Theta(1)} \Rightarrow \sum_{i=1}^{n} f(i) = \Theta(1)$

Use as thmbrules only