2301520 FUNDAMENTALS OF AMCS

Lecture 2: Complexity

Lectured by Dr. Krung Sinapiromsaran,
Krung.S@chula.ac.th

Excerpted from Dr. William Smith,
wsmith@cs.york.ac.uk

Outline

- Algorithm performance
- Grouping inputs by size
- Worst-case, best-case and average-case analysis
- Measuring resource usage
- RAM model of computation
- Asymptotic notation:Big Oh, Big Omega, Theta, little oh, little omega
- Complexity usages

Objective

- We study how we analyze an algorithm.
- To compare several algorithms that solve the same problem, we group inputs by their sizes.
- Three types of analysis are measured. All are based on RAM model.
- We introduce an Asymptotic notation, O, Ω, Θ, o, ω.
- Then we apply this to classify different Algorithm class

Algorithm performance (1)

Q: How might we establish whether algorithm A is faster than algorithm B?

Algorithm performance (2)

Q: How might we establish whether algorithm A is faster than algorithm B?

A1: We could implement both of them, run them on the same input and time how long each of them takes

Algorithm performance (3)

Q: How might we establish whether algorithm A is faster than algorithm B?

A1: We could implement both of them, run them on the same input and time how long each of them takes

- Unfair test: what if one of the algorithms just happens to be faster on this particular input?

Algorithm performance (4)

Q: How might we establish whether algorithm A is faster than algorithm B?

A2: We could implement both of them, run them on lots of different inputs and time how long each of them takes on each input

Algorithm performance (5)

Q: How might we establish whether algorithm A is faster than algorithm B?

A2: We could implement both of them, run them on lots of different inputs and time how long each of them takes on each input

- Assuming we can try every input of a particular size, this would give us best, worst and average running times for this particular implementation on this particular computer for this particular input size
- Still an unfair test: what if one algorithm just happens to be faster on this size of input?
- What if we want a more general answer? Not tied to one computer or implementation.

Algorithm performance (6)

Let's generalise things slightly…

The function: $T: I \rightarrow R^+$

is a mapping from the set of all inputs I to the time taken on that input

- For any problem instance *i* in I, T(*i*) is the running time on *i*.
  - Computing the running time for every possible problem instance is overwhelming
  - Instead, group together "similar" inputs
  - Gives us running time as a function of a class of instances
  - How shall we group inputs?

Grouping inputs by size (1)

Grouping inputs together of equal size is generally the most useful

Bigger problems are harder to solve

Q:What do we mean by the size of an input?

Grouping inputs by size (2)

Grouping inputs together of equal size is generally the most useful

Bigger problems are harder to solve

Q:What do we mean by the size of an input?

A:It depends on the problem.

Grouping inputs by size (3)

Grouping inputs together of equal size is generally the most useful

Bigger problems are harder to solve

Q:What do we mean by the size of an input?

A:It depends on the problem.

- Integer input → number of digits
- Set input → number of elements in a set
- Text string → number of characters

- Generally obvious

Grouping inputs by size (4)

Grouping inputs together of equal size is generally the most useful

Bigger problems are harder to solve

Q:What do we mean by the size of an input?

A:It depends on the problem.

- Integer input → number of digits
- Set input → number of elements in a set
- Text string → number of characters

- Generally obvious

Not always so neat: what if the input was a graph?

May need more than one size parameter: graph size = (# vertices, # edges)

Types of performance analysis (1)

We denote the set of all instances of size n in N as I_n.

We can define three measures of performance:

Types of performance analysis (2)

We denote the set of all instances of size n in N as I_n.

We can define three measures of performance:

- Worst-case: $T(n) = \max\{T(i) \mid i \text{ in } I_n\}$

  $T(n)$ = maximum time of algorithm on any input of size n.

Types of performance analysis (3)

We denote the set of all instances of size n in N as I_n.

We can define three measures of performance:

- Worst-case: $\mathrm{T}(n) = \max\{\mathrm{T}(i) \mid i \text{ in } I_n\}$

  $\mathrm{T}(n)$ = maximum time of algorithm on any input of size n.

- Best-case: $\mathrm{T}(n) = \min\{\mathrm{T}(i) \mid i \text{ in } I_n\}$

  $\mathrm{T}(n)$ = minimum time of algorithm on any input of size n.

Types of performance analysis (4)

We denote the set of all instances of size n in N as I_n.

We can define three measures of performance:

- Worst-case: $\mathrm{T}(n) = \max\{\mathrm{T}(i) \mid i \text{ in } I_n\}$

  $\mathrm{T}(n)$ = maximum time of algorithm on any input of size n.

- Best-case: $\mathrm{T}(n) = \min\{\mathrm{T}(i) \mid i \text{ in } I_n\}$

  $\mathrm{T}(n)$ = minimum time of algorithm on any input of size n.

- Average-case: $\mathrm{T}(n) = \dfrac{1}{|I_n|}\sum_{i \in I_n} T(i)$

  $\mathrm{T}(n)$ = expected time of algorithm on any input of size n.

Types of performance analysis (5)

We denote the set of all instances of size n in N as I_n.

We can define three measures of performance:

- Worst-case: $T(n) = \max\{T(i) \mid i \text{ in } I_n\}$

  T(n) = maximum time of algorithm on any input of size n.

- Best-case: $T(n) = \min\{T(i) \mid i \text{ in } I_n\}$

  T(n) = minimum time of algorithm on any input of size n.

- Average-case: $T(n) = \frac{1}{|I_n|}\sum_{i \in I_n} T(i)$

  T(n) = expected time of algorithm on any input of size n.

Q:What assumption is being made here?

Types of performance analysis (6)

We denote the set of all instances of size n in N as I_n.

We can define three measures of performance:

- Worst-case: $T(n) = \max\{T(i) \mid i \text{ in } I_n\}$

  $T(n)$ = maximum time of algorithm on any input of size n.

- Best-case: $T(n) = \min\{T(i) \mid i \text{ in } I_n\}$

  $T(n)$ = minimum time of algorithm on any input of size n.

- Average-case: $T(n) = \frac{1}{|I_n|}\sum_{i \in I_n} T(i)$

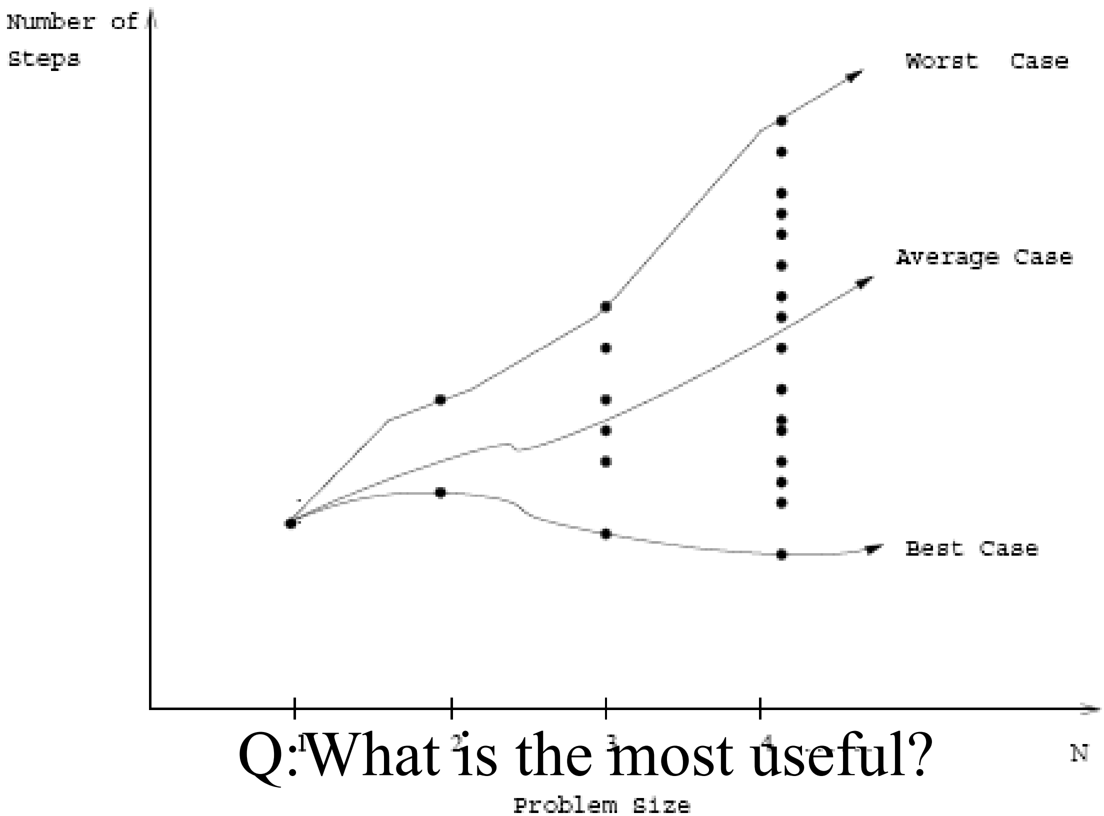
  $T(n)$ = expected time of algorithm on any input of size n.

Q:What assumption is being made here?

All inputs equally likely – if not we need to know the probability distribution

Types of performance analysis (7)

We denote the set of all instances of size n in N as I_n.
We can define three measures of performance:



Q:What is the most useful?

Q:How can we modify almost any algorithm to have a good best case running time?

Types of performance analysis (8)

Q: Which is most useful?

A: Generally concentrate on worst-case execution time – strongest performance guarantee

Types of performance analysis (9)

Q: Which is most useful?

A: Generally concentrate on worst-case execution time – strongest performance guarantee

Q: How can we modify almost any algorithm to have a good best-case running time?

A: Find a solution for one particular input and store it. When that input is encountered, return our precomputed answer immediately.

Other more subtle ways of improving best-case performance.

Best-case is generally bogus!

Measuring resource usage (1)

Example

Summing the first n positive integers:

Precondition: n in N; Postcondition: $r = \sum_{i=1}^{n} i$

Two solutions:

```
r := 0
for i := 1 to n do
    r := r + i
endfor
```

$$r := \frac{n(n+1)}{2}$$

Measuring resource usage (2)

Example

Summing the first n positive integers:

Precondition: n in N; Postcondition: $r = \sum_{i=1}^{n} i$

Two solutions:

```
r := 0
for i := 1 to n do
    r := r + i
endfor
```

$$r := \frac{n(n+1)}{2}$$

Both algorithms are correct.

Q: Which is better?

Measuring resource usage (3)

Define some constants:

- i is the time to increment by 1
- a is the time to perform an addition
- t is the time to perform the loop test
- m is the time to multiply two numbers
- d is the time to divide by 2
- s is the time to perform an assignment

Measuring resource usage (4)

Define some constants:

- *i* is the time to increment by 1
- *a* is the time to perform an addition
- *t* is the time to perform the loop test
- *m* is the time to multiply two numbers
- *d* is the time to divide by 2
- *s* is the time to perform an assignment

Version 1	**Cost**	**Number of Times:**

```
r := 0
for i := 1 to n do
    r := r + i
endfor
```

Measuring resource usage (5)

Define some constants:

- *i* is the time to increment by 1
- *a* is the time to perform an addition
- *t* is the time to perform the loop test
- *m* is the time to multiply two numbers
- *d* is the time to divide by 2
- *s* is the time to perform an assignment

Version 1	**Cost**	**Number of Times:**
r := 0	*s*	1
for i := 1 to n do		
r := r + i		
endfor		

Measuring resource usage (6)

Define some constants:

- i is the time to increment by 1
- a is the time to perform an addition
- t is the time to perform the loop test
- m is the time to multiply two numbers
- d is the time to divide by 2
- s is the time to perform an assignment

Version 1	**Cost**	**Number of Times:**
r := 0	s	1
for i := 1 to n do	$t + i$	n+1
r := r + i		
endfor		

Measuring resource usage (7)

Define some constants:

- i is the time to increment by 1
- a is the time to perform an addition
- t is the time to perform the loop test
- m is the time to multiply two numbers
- d is the time to divide by 2
- s is the time to perform an assignment

Version 1	**Cost**	**Number of Times:**
r := 0	s	1
for i := 1 to n do	$t + i$	n+1
r := r + i	$a + s$	n
endfor		

Measuring resource usage (8)

Define some constants:

- i is the time to increment by 1
- a is the time to perform an addition
- t is the time to perform the loop test
- m is the time to multiply two numbers
- d is the time to divide by 2
- s is the time to perform an assignment

Version 1	**Cost**	**Number of Times:**
r := 0	s	1
for i := 1 to n do	$t + i$	n+1
r := r + i	$a + s$	n
endfor	$T_1 = n(t+i+a+s) + t+i + s$	

Measuring resource usage (9)

Define some constants:

- i is the time to increment by 1
- a is the time to perform an addition
- t is the time to perform the loop test
- m is the time to multiply two numbers
- d is the time to divide by 2
- s is the time to perform an assignment

Version 2 **Cost** **Number of Times:**

$$r := \frac{n(n+1)}{2}$$

Measuring resource usage (10)

Define some constants:

- i is the time to increment by 1
- a is the time to perform an addition
- t is the time to perform the loop test
- m is the time to multiply two numbers
- d is the time to divide by 2
- s is the time to perform an assignment

Version 2	**Cost**	**Number of Times:**
$r := \frac{n(n+1)}{2}$	$i+m+d+s$	1

Measuring resource usage (11)

Define some constants:

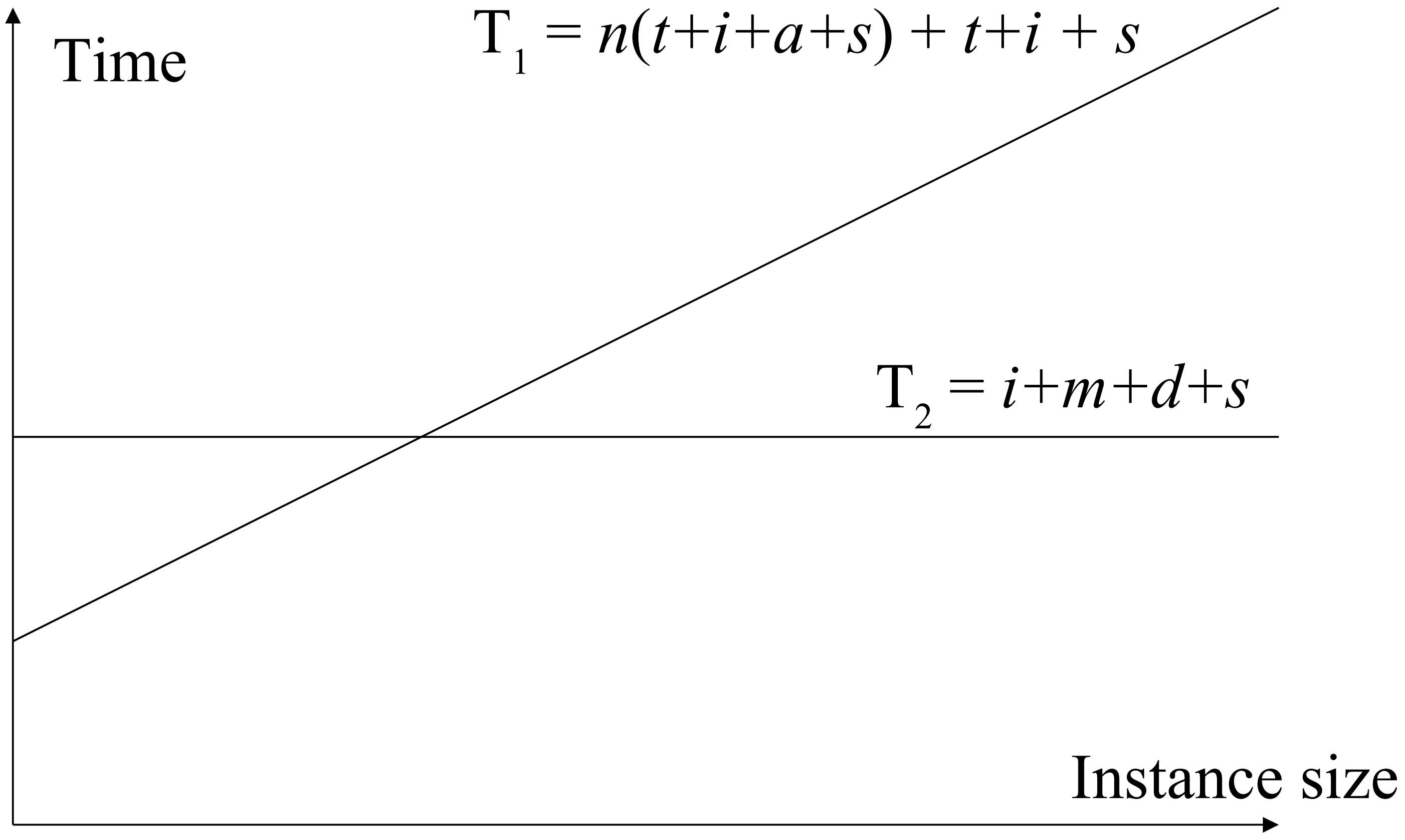
- i is the time to increment by 1
- a is the time to perform an addition
- t is the time to perform the loop test
- m is the time to multiply two numbers
- d is the time to divide by 2
- s is the time to perform an assignment

Version 2	**Cost**	**Number of Times:**
$r := \frac{n(n+1)}{2}$	$i+m+d+s$	1

$$T_2 = i+m+d+s$$

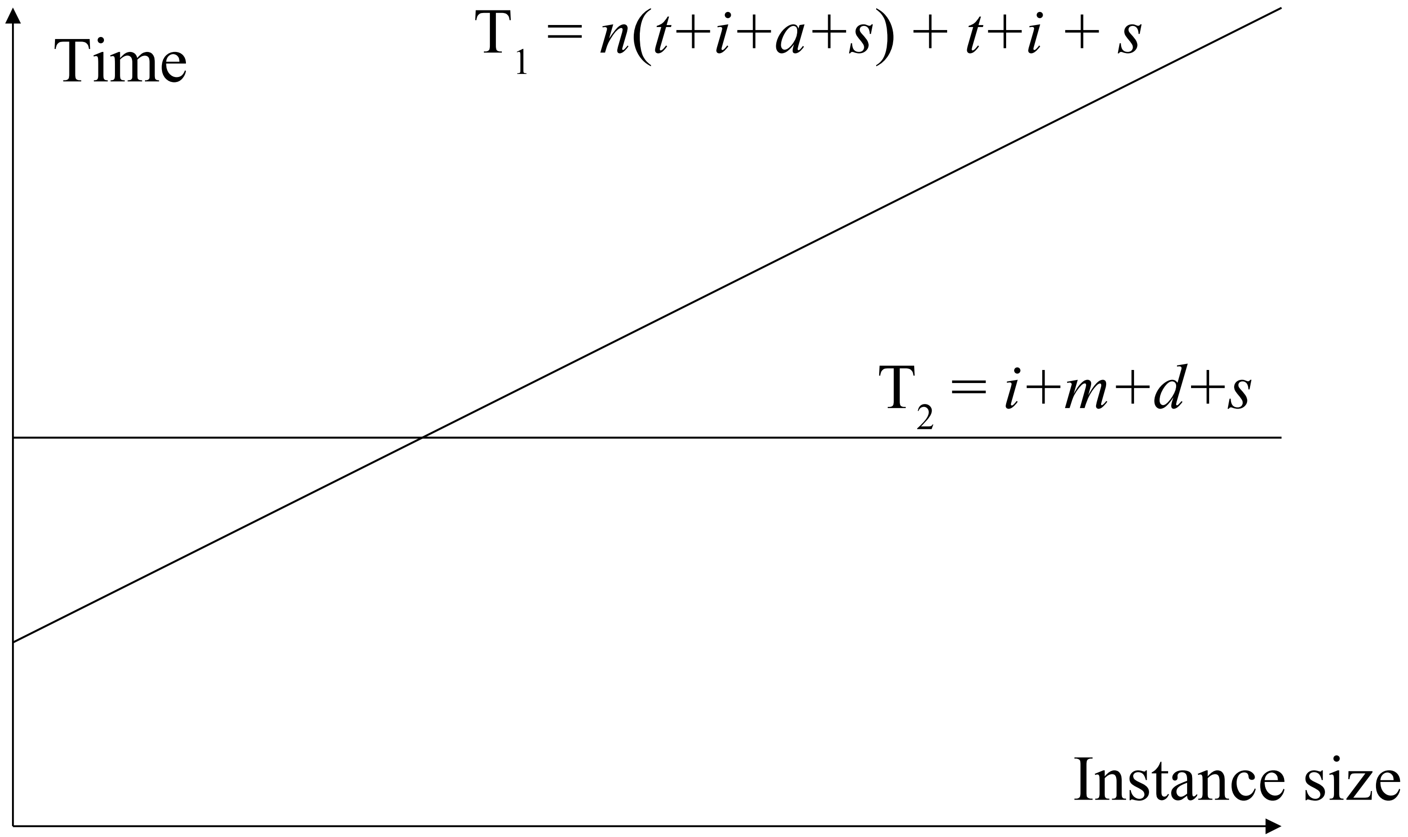
Measuring resource usage (12)

Which is better?



Measuring resource usage (13)

Which is better?



Depends on size of input. Beyond intersection T_2 will always win

The RAM model of computation

- The above analysis made some implicit assumptions
- Modern hardware is hugely complex (pipelines, multiple cores, caches etc)
- We need to abstract away from this
- We require a model of computation that is simple and machine independent
- Typically use a variant of a model developed by John von Neumann in 1945
- Programs written with his model in mind run efficiently on modern hardware

Operations on RAM model

- Each simple operation (+, *, -, =, if, assignment) takes exactly one time step
- Loops and subroutine calls not considered simple operations
- We have a finite, but always sufficiently large, amount of memory
- Each memory access takes exactly one time step
- Instructions are executed one after another
- Time $\propto$ number of instructions

Exact analysis is hard!

- RAM model justifies counting number of operations in our algorithms to measure execution time.
- Only predict real execution times up to a constant factor
- Precise details depend on uninteresting coding details
- Constant speedups just reflect running code on a faster computer
- We are really interested in machine independent growth rates
- Why?
- We are interested in performance for large n, we want to be able to solve difficult instances; start-up time dominates for small n
- Known as ***asymptotic analysis***
- We can characterize and compare running times of algorithms with simple functions

Asymptotic Notation

- Consider two functions $f(n)$ and $g(n)$ with integer inputs and numerical outputs
- We say f grows no faster then g in the limit if:

There exist positive constants c and n_0 such that

$$f(n) \leq c\, g(n) \text{ for all } n > n_0$$

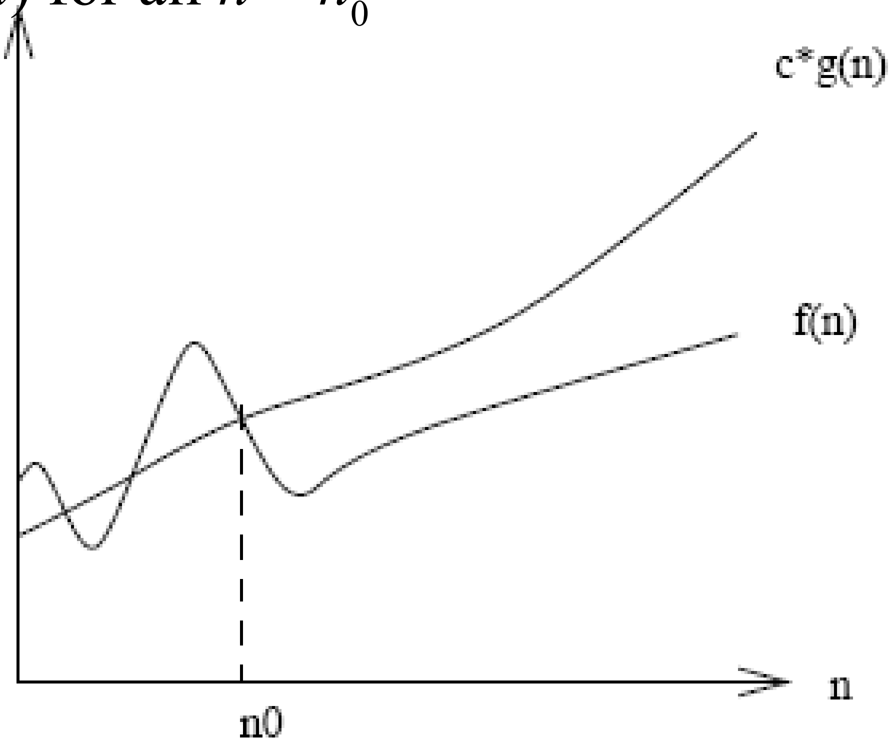
We write this as: $f(n) = O(g(n))$

Read as "f is Big Oh of g"

We can also say "f is asymptotically dominated by g"

"g is an upper bound on f"

"f grows no faster than g"



Definition of Big Oh (1)

- Format definition:

$$f(n) = O(g(n)) \text{ iff } \exists c \in R^+; n_0 \in N, \forall n > n_0, f(n) \leq c\, g(n)$$

- Breaking this up:

n_0, $n > n_0$ means we don't care about small n.

c, $f(n) \leq c\, g(n)$ means we don't care about constant speedups.

Unusual notation: "one way equality"

Really an ordering relation (think of < and >)

$f(n) = O(g(n))$ definitely does not imply $g(n) = O(f(n))$

Definition of Big Oh (2)

- Might like to think in terms of sets:

$$O(g(n)) = \{f(n) \mid \exists c \in R^+; n_0 \in N, \forall n > n_0, f(n) \leq c\, g(n)\}$$

- In this way:we can interpret $f(n) = O(g(n))$ as $f(n) \in O(g(n))$

Sometimes read as "*f* is in Big Oh of *g*"

Big Oh example (1)

$n^2 + 1 = O(n^2)$ – True or false? --------(*)

How would we prove it?

- Consider the definition:

  $$f(n) = O(g(n)) \text{ iff } \exists c \in R^+; n_0 \in N, \forall n > n_0, f(n) \leq c\, g(n)$$

- To prove $\exists x$P we need:
  - A witness (value) for x
  - A proof that P holds when witness substituted for x.

Big Oh example (2)

$$n^2 + 1 = O(n^2) \text{ – True -------(*)}$$

- Let choose $c = 2$
- Need to find an n_0 such that

$$\forall n > n_0,\ n^2 + 1 \leq 2n^2$$

- In this case, $n_0 = 1$ or greater value will do.
- By convention, always complex to simple:

$$\text{complex} = O(\text{simple})$$

- e.g. $3n^2 + 102n + 56 = O(n^2)$

  $3n^2 + 102n + 56 = O(n^3)$

  $3n^2 + 102n + 56 = O(n)$
- Related operators follow from definition of Big Oh…

Definition of Big Omega

- If Big Oh is like $\leq$ then Big Omega is like $\geq$
- "f grows no slower than g"

$$f(n) = \Omega(g(n)) \text{ iff } g(n) = O(f(n))$$

- Read as "f is Big Omega of g"
- Express as a set:

$$\Omega(g(n)) = \{f(n) \mid \exists c \in R^+; n_0 \in N,\ \forall n > n_0,\ f(n) \geq c\, g(n)\}$$

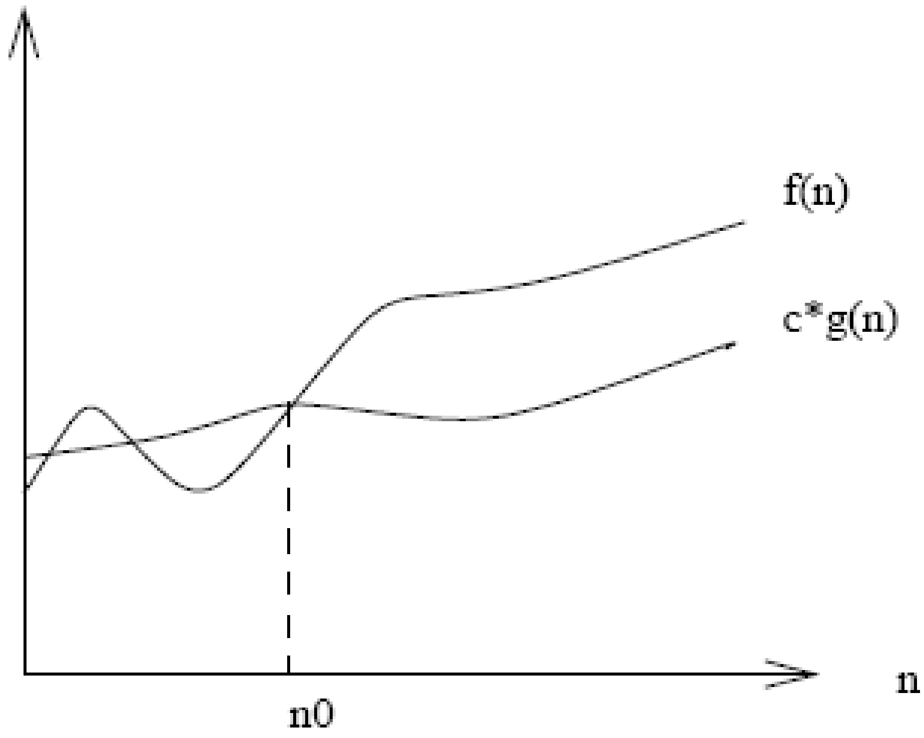
Same as Big Oh just reverse equality

- e.g. $3n^2 + 102n + 56 = \Omega(n^2)$

  $3n^2 + 102n + 56 = \Omega(n^3)$

  $3n^2 + 102n + 56 = \Omega(n)$

Graph of Big Omega



Definition of Big Theta

- Big Theta is like =
- "*f* grows at the same rate as *g*"

$$f(n) = \Theta(g(n)) \text{ iff } f(n) = O(g(n)) \ \& \ f(n) = \Omega(g(n))$$

- Read as "*f* is Big Theta of *g*"
- Express as a set:

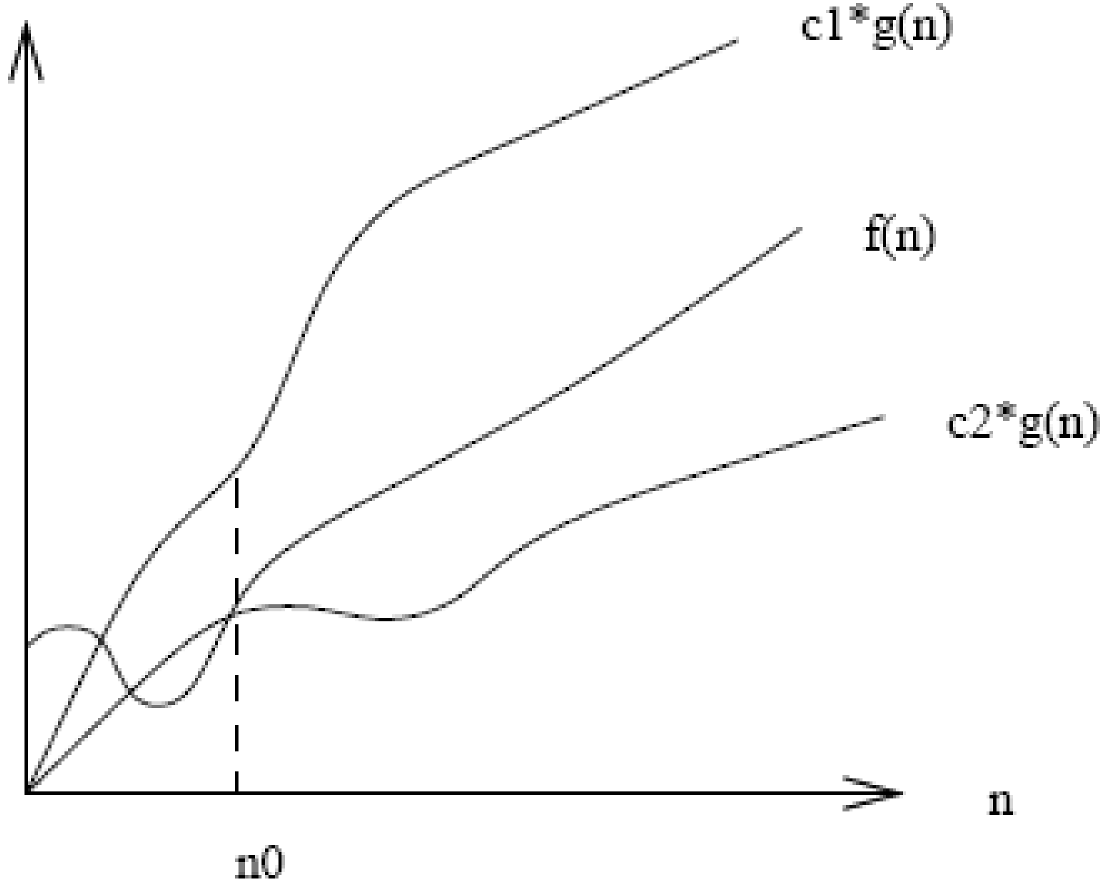
$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$

- e.g. $3n^2 + 102n + 56 = \Theta(n^2)$

  $3n^2 + 102n + 56 = \Theta(n^3)$

  $3n^2 + 102n + 56 = \Theta(n)$

Graph of Big Theta



Definition of little oh

- If Big Oh is like $\leq$ then little oh is like $<$
- "f grows strictly slower than g"

$$f(n) = o(g(n)) \text{ iff } f(n) = O(g(n)) \;\&\; f(n) \neq \Theta(g(n))$$

- Read as "f is little oh of g"
- Express as a set:

$$o(g(n)) = \{f(n) \mid \forall c \in \mathbb{R}^+; n_0 \in \mathbb{N}, \forall n > n_0, f(n) < c\, g(n)\}$$

Same as Big Oh, but existential becomes universal

- e.g. $3n^2 + 102n + 56 = o(n^2)$
  $3n^2 + 102n + 56 = o(n^3)$
  $3n^2 + 102n + 56 = o(n)$

Definition of little omega

- If Big Omega is like $\geq$ then little omega is like $>$
- "f grows strictly faster than g"

$$f(n) = \omega(g(n)) \text{ iff } f(n) = \Omega(g(n)) \ \& \ f(n) \neq \Theta(g(n))$$

- Read as "f is little omega of g"
- Express as a set:

$$\omega(g(n)) = \{f(n) \mid \forall c \in \mathrm{R}^+; n_0 \in \mathrm{N}, \forall n > n_0, f(n) > c\ g(n)\}$$

Same as Big Omega, but existential becomes universal

- e.g. $3n^2 + 102n + 56 = \omega(n^2)$

$3n^2 + 102n + 56 = \omega(n^3)$

$3n^2 + 102n + 56 = \omega(n)$

Summary

$\leq$	$f(n) = O(g(n))$ iff $\exists c \in R^+; n_0 \in N, \forall n > n_0, f(n) \leq c\, g(n)$
$\geq$	$f(n) = \Omega(g(n))$ iff $g(n) = O(f(n))$
$=$	$f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
$<$	$f(n) = o(g(n))$ iff $f(n) = O(g(n))$ and $f(n) \neq \Theta(g(n))$
$>$	$f(n) = \omega(g(n))$ iff $f(n) = \Omega(g(n))$ and $f(n) \neq \Theta(g(n))$

An alternative limit-based interpretation:

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

$$f(n) := \omega(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

$$f(n) := \Theta(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = r > 0$$

- Q: How might we use this to empirically test the complexity of an algorithm implementation?

Practical complexity theory (1)

- Properties of Big Oh and others leads to mechanical rules for simplification
- Drop low order terms
- Ignore leading constants

$3n^3 + 90n^2 + 5\text{n} + 6046$

Practical complexity theory (2)

- Properties of Big Oh and others leads to mechanical rules for simplification
- Drop low order terms
- Ignore leading constants

$3n^3 + \cancel{90}n^2 + \cancel{5}n + \cancel{60}46$

Practical complexity theory (3)

- Properties of Big Oh and others leads to mechanical rules for simplification
- Drop low order terms
- Ignore leading constants

$\cancel{3}n^3 + \cancel{90n^2} + \cancel{5n} + \cancel{6046}$

Practical complexity theory (4)

- Properties of Big Oh and others leads to mechanical rules for simplification
- Drop low order terms
- Ignore leading constants

$$\cancel{3}n^3 + \cancel{90n^2} + \cancel{5n} + \cancel{6046} = \Theta(n^3)$$

Conclusion

- We now have some tools for algorithm analysis allowing us to talk abstractly about the complexity of an algorithm.
- Next, we will learn how to apply this tool
- Classify the complexity class
- Which level of complexity is considered “efficient” or “do-able”?