Analysis of Algorithm and Design
Lecture 1

Acknowledgement

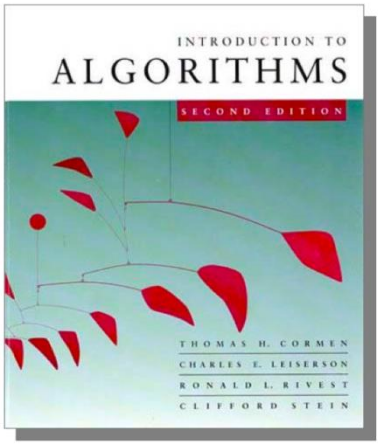
- This lecture note has been summarized from lecture note on Data Structure and Algorithm, Design and Analysis of Computer Algorithm all over the world. I can't remember where those slide come from. However, I'd like to thank all professors who create such a good work on those lecture notes. Without those lectures, this slide can't be finished.

More Information

❖Textbook

- ***Introduction to Algorithms 2nd*** ,*Cormen, Leiserson, Rivest and Stein*, The MIT Press, 2001.
- ***Data Structures and Algorithm Analysis***

Clifford A. Shaffer,Edition 3.2 (C++ Version),2012.



❖Others

- ***Introduction to Design & Analysis Computer Algorithm 3rd,*** Sara Baase, Allen Van Gelder, Adison-Wesley, 2000.
- ***Algorithms***, Richard Johnsonbaugh, Marcus Schaefer, Prentice Hall, 2004.
- ***Introduction to The Design and Analysis of Algorithms 2nd Edition,*** Anany Levitin, Adison-Wesley, 2007.

Course Objectives

- This course introduces students to the analysis and design of computer algorithms. Upon completion of this course, students will be able to do the following:
  - Analyze the asymptotic performance of algorithms.
  - Demonstrate a familiarity with major algorithms and data structures.
  - Apply important algorithmic design paradigms and methods of analysis.
  - Synthesize efficient algorithms in common engineering design situations.

Objective of course

In this course we will cover the following topics:

- Understand foundations of algorithms && design and analysis various variants algorithms

  Accuracy

  Efficiency

  Comparing efficiencies

- Make use of skills to understand mathematical notations in algorithms and their simple mathematical proofs

- Gain familiarity with a number of classical problems that occur frequently in real-world applications

Sunday, July 6, 2025

COURSE OUTLINE

- **Preliminaries** : different types of algorithms, analyzing methodologies, notations, proof techniques and limits.
- **Asymptotic Notation** : different notations and their examples, standard notations and their common functions.
- **Analysis of Algorithms** : analyzing control structures, using barometer instruction, amortization, and different examples for analysis and solving recurrences.
- **Structures**: use of arrays, stacks, queues, records, pointers, lists, graphs, trees, hash tables, heaps and binomial heaps.
- **Searching/Sorting Algorithms** : Various searching and sorting algorithms and their comparisons.

COURSE OUTLINE

- Tree Algorithms: understanding and making different types of trees and their algorithms.
- Graph Algorithms: understanding and making different types of graphs and their algorithms for finding shortest paths
- Heap Algorithms : developing heaps, binomial heaps, disjoint structures, and their algorithms.
- Divide and Conquer: Multiplication of Integers, Binary Search, Sorting Algorithms and Matrix Multiplication.
- Greedy Algorithms : General Characteristics of Greedy Algorithms, Minimum Spanning Trees, Shortest Graph Paths, Huffman Codes and Scheduling.
- NP-Completeness: understanding of P, NP and NP-completeness and use of algorithms for some problems

What is Algorithm?

- Algorithm
  - is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.
  - is thus a sequence of computational steps that transform the input into the output.
  - is a tool for solving a well - specified computational problem.
  - Any special method of solving a certain kind of problem (Webster Dictionary)

What is the specification of algorithm?

- We can specify an algorithm by 3 types:
- 1. Using Natural language
- 2. Pseudocode;
- 3. And flowchart.

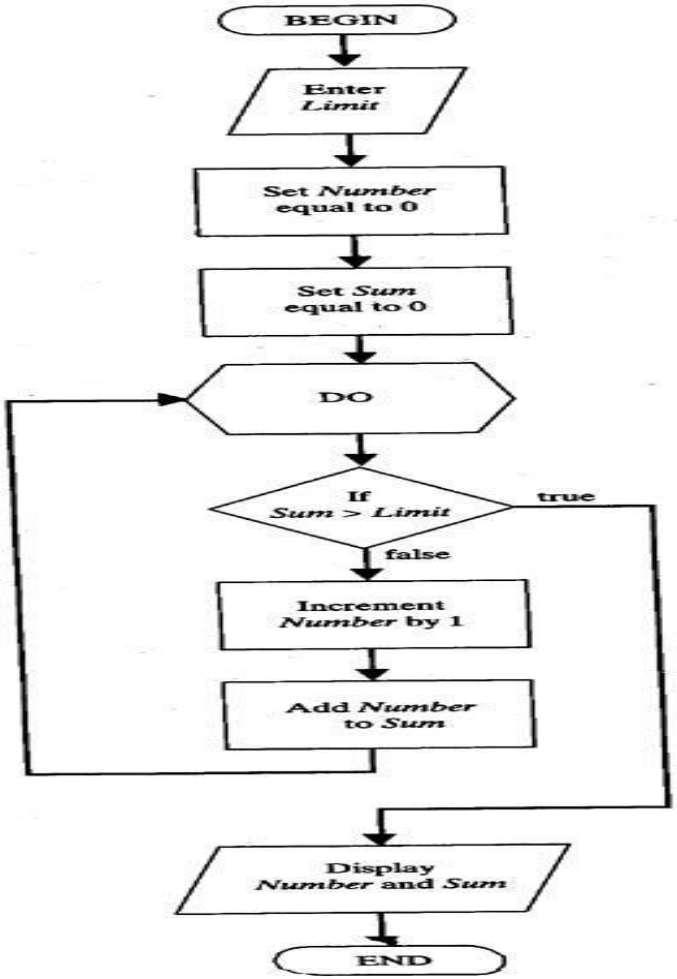
- Natural language is general way to solve any problem
- For e.g to perform addition of two numbers we write in natural language
- Step1: read the first number (assume a)
- Step2:read the second number (assume b)
- Step3:add 2 numbers(a and b)and store the result in a variable c
- Step4:display the result

Pseudocode

Combination of natural language and programming language construct For e.g.to perform addition of two numbers we write in pseudo-code

- //this algo perform addition of two integers
- //input two integers a and b
- // output of two integers
- C=a+b
- Write c

Another way to represent an algo is Flow chart .



ALGORITHMICS

It is the science that lets designers study and evaluate the effect of algorithms based on various factors so that the best algorithm is selected to meet a particular task in given circumstances. It is also the science that tells how to design a new algorithm for a particular job.

What is a program?

- A program is the expression of an algorithm in a programming language
- a set of instructions which the computer will follow to solve a problem

Analysis of an Algorithm

- **What is analysis of an algorithm?**
- **Ans.** Algorithm analysis is an important part of a computational complexity theory.
- There two types of analysis:
- 1.priori
- 2.posteriory

Analysis of an Algorithm

- Priori Analysis is the theoretical estimation of resources required.

- On the other hand posterior Analysis done after implement the algorithm on a target machine.

Introduction

- Why need algorithm analysis ?
  - writing a working program is not good enough
  - The program may be inefficient!
  - If the program is run on a large data set, then the running time becomes an issue

Complexity of an algorithm

- **What is the complexity of an algorithm?**
- Complexity (or efficiency) of an algorithm is a function that describes the efficiency of an algorithm in terms of time and space.
- There are two types of complexity:
- 1.Time complexity:
- Time complexity means the amount of time required by an algorithm to run.
- It expressed as an order of magnitude / degree

❖2.space complexity:

- Space complexity means the amount of space required by an algorithm to run.It expressed as an order of magnitude.There are two important factors to compute the space complexity:constant and instant characterstic.
- e.g
- factorial(n)
- if n=0 then
- return 1;
- else
- return*factorial(n-1);
- endif
- space complexity of program = constant size + variable size

Types of Algorithms

PROBABILISITIC ALGORITHM

- In this algorithm, chosen values are used in such a way that the probability of chosen each value is known and controlled.

e.g. Randomize Quick Sort

HEURISTIC ALGORITHM

- **This type of algorithm is based largely on optimism and often with minimal theoretical support. Here error can not be controlled but may be estimated how large it is.**

APPROXIMATE ALGORITHM

In this algorithm, answer is obtained that is as précised as required in decimal notation. In other words it specifies the error we are willing to accept.

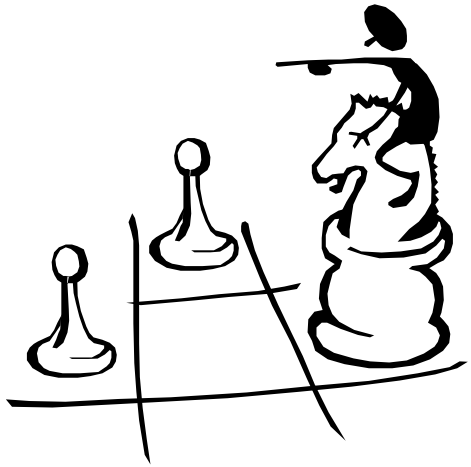
For example, two figures accuracy or 8 figures or whatever is required.

Things to do……

- Learn general approaches to algorithm design
  - Divide and conquer
  - Greedy method
  - Dynamic Programming
  - Basic Search and Traversal Technique
  - Graph Theory
  - Linear Programming
  - Approximation Algorithm
  - NP Problem

Some Application

- Study problems these techniques can be applied to
  - sorting
  - data retrieval
  - network routing
  - Games
  - etc



What do we analyze about them?

- Correctness
  - Does the input/output relation match algorithm requirement?
- Amount of work done (aka complexity)
  - Basic operations to do task finite amount of time
- Amount of space used
  - Memory used

Strengthening the Informal Definition

- Important Features:
  - Finiteness.[algo should end in finite amount of steps]
  - Definiteness.[each instruction should be clear]
  - Input.[valid input clearly specified]
  - Output.[single/multiple valid output]
  - Effectiveness.[steps are sufficiently simple and basic]

performance analysis

- **What is performance analysis?**
- Performance analysis is the criteria for judging the algorithm.It have direct relationship to performance.When we solve a problem ,there may be more then one algorithm to solve a problem,through analysis we find the run time of an algorithms and we choose the best algorithm which takes lesser run time.

Example Of Algorithm

Few Classical Examples

Classical Multiplication Algorithms

- English
- American
- A la russe
- Divide and Conquer

Classic Multiplication

CLASSIC MULTIPLICATION ALGORITHMS

(981 x 1234)

981	981
1234	1234
3924	981
2943	1962
1962	2943
981	3924
1210554	1210554
American	English

Multiplication

MULTIPLICATION (981 x 1234)

(*a la russe algorithm*)

981	1234	1234
490	2468	
245	4936	4936
122	9872	
61	19744	19744
30	39488	
15	78976	78976
7	157952	157952
3	315904	315904
1	631808	631808
		1210554

Multiplication

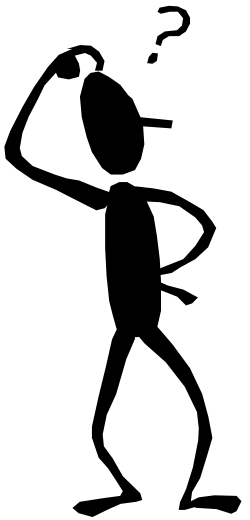
MULTIPLICATION (981 x 1234)

(*Divide-and-Conquer Algorithm*)

	Multiply		Shift	Result
i)	09	12	4	108….
ii)	09	34	2	306..
iii)	81	12	2	972..
iv)	81	34	0	2754
				1210554

The Selection Problem

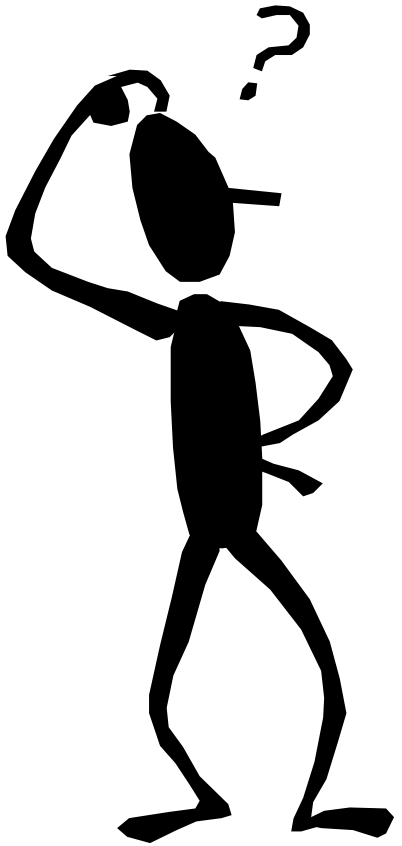
Which algorithm is better?



Which algorithm is better?

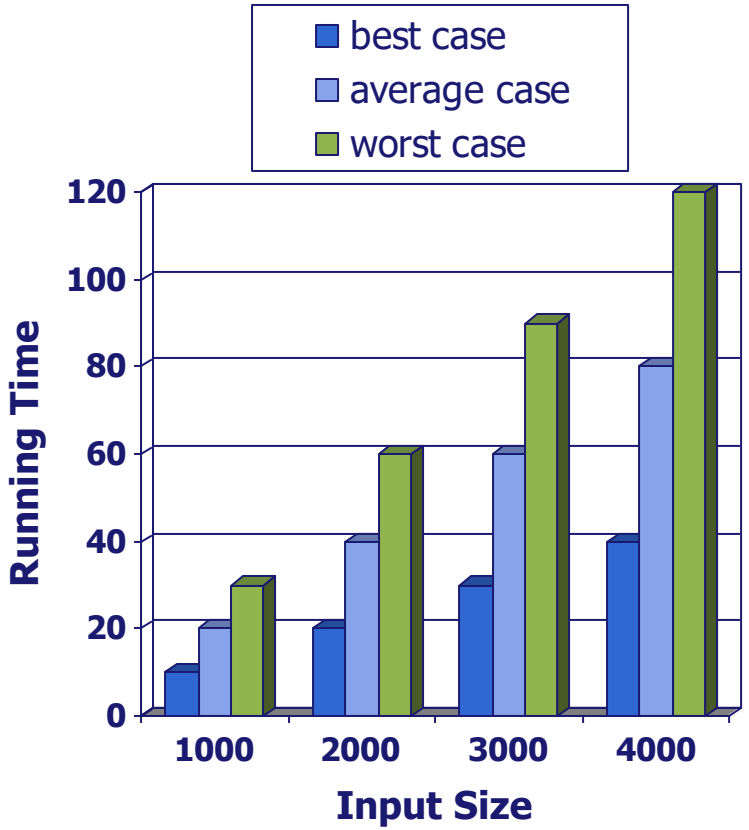
The algorithms are correct, but which is the best?

- Measure the running time (number of operations needed).
- Measure the amount of memory used.
- Note that the running time of the algorithms increase as the size of the input increases.



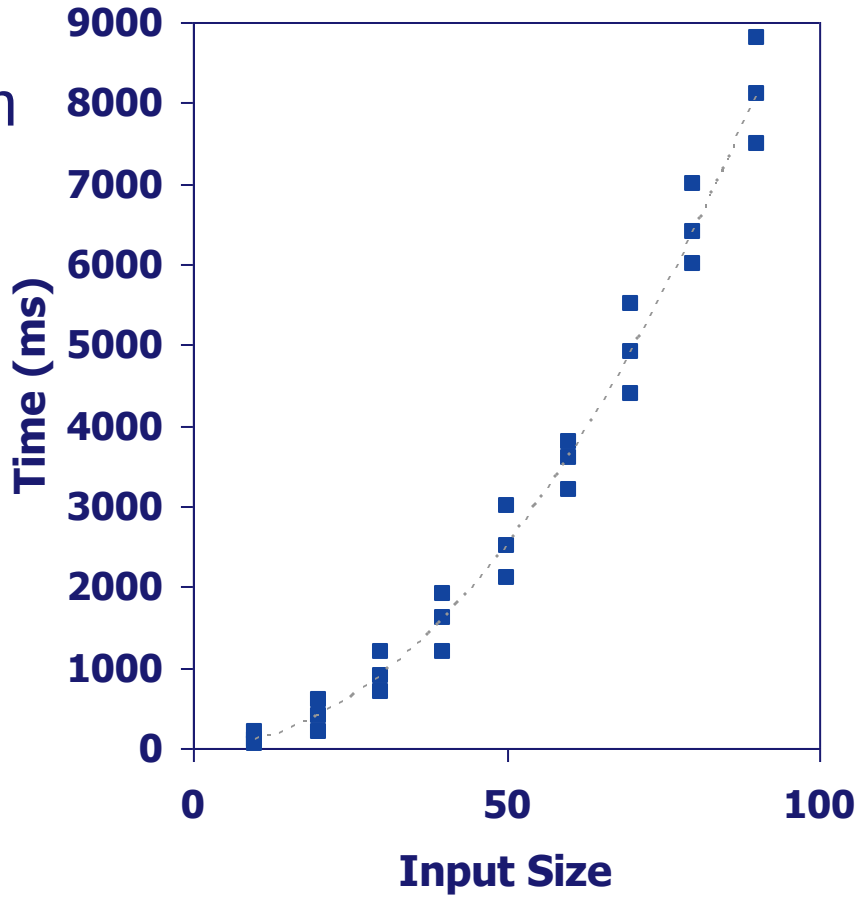
Running Time

- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- We focus on the worst case running time.
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics



Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like System.currentTimeMillis() to get an accurate measure of the actual running time
- Plot the results



Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used

Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, *n*.
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

RAM model

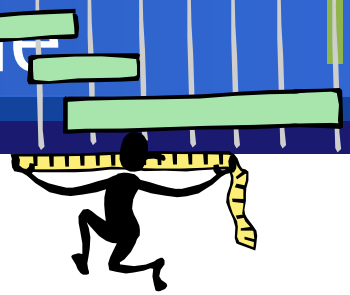
- has one processor
- executes one instruction at a time
- each instruction takes "unit time“
- has fixed-size operands, and
- has fixed size storage (RAM and disk).

Counting Primitive Operations

- By inspecting the pseudo code, we can determine the maximum number of primitive/basic operations executed by an algorithm, as a function of the input size

Algorithm ***arrayMax*(*A*, *n*)**	# operations
currentMax ← ***A*[0]**	2
for (***i*** =1; i<n; i++)	2***n***
(i=1 once, i<n n times, i++ (n-1) times:post increment i.e 1+ n + n -1 = 2n)	
if ***A*[*i*] > *currentMax*** **then**	2(***n*** – 1)
currentMax ← ***A*[*i*]**	2(***n*** – 1)
return ***currentMax***	1
Total	6***n*** –1

Estimating Running Time



- Algorithm ***arrayMax*** executes $6n - 1$ primitive operations in the worst case.

  Define:

  a = Time taken by the fastest primitive operation

  b = Time taken by the slowest primitive operation

- Let $T(n)$ be worst-case time of ***arrayMax***. Then

$$a\,(6n - 1) \leq T(n) \leq b(6n - 1\,)$$

- Hence, the running time $T(n)$ is bounded by two linear functions

Growth Rate of Running Time

- Changing the hardware/ software environment
  - Affects $T(n)$ by a constant factor, but
  - Does not alter the growth rate of $T(n)$
- The linear growth rate of the running time $T(n)$ is an essential property of algorithm ***arrayMax***

Function of Growth rate

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	N log N
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Functions in order of increasing growth rate

Big-Oh Notation

- To simplify the running time estimation, for a function *f(n)*, we ignore the constants and lower order terms.

Example: $10n^3+4n^2-4n+5$ is $O(n^3)$.

Why ???

- Why we are measuring the complexity of Algorithms ???.

- Reason : Not only to design algorithms but to design efficient type of algorithms which can finish in finite amount of time.

Why asymptotic Notations

- Asymptotic notation is a way to represent the time complexity
- Design efficient algorithm terminate in finite amount of Time/ acceptable amount of time.

Making use of Asymptotic notations for measuring running time of algorithm for large input size and measure its growth rate .

Input Size

- Input size (number of elements in the input)
  - size of an array
  - polynomial degree
  - # of elements in a matrix
  - # of bits in the binary representation of the input
  - vertices and edges in a graph

Efficiency Measurement

- Efficiency relative term
- Single algorithm : Polynomial Time Period (Efficient)
- Comparing Algorithms for finding their efficiencies

Types of Analysis

- Worst case (at most BIG O)
  - Provides an upper bound on running time
  - An absolute guarantee that the algorithm would not run longer, no matter what the inputs are
- Best case (at least Omega Ω)
  - Provides a lower bound on running time
  - Input is the one for which the algorithm runs the fastest

$$Lower\ Bound \leq Running\ Time \leq Upper\ Bound$$

- Average case (Theta Θ)
  - Provides a prediction about the running time
  - Assumes that the input is random

How do we compare algorithms?

- We need to define a number of objective measures.

(1) Compare execution times?

Not good: times are specific to a particular computer !!

(2) Count the number of statements executed?

Not good: number of statements vary with the programming language as well as the style of the individual programmer.

Ideal Solution

- Express running time as a growth function of the input size *n* (i.e., *f(n)*).
- Compare different functions corresponding to running times.
- Such an analysis is independent of machine time, programming style, etc.

Asymptotic Analysis

- To compare two algorithms with running times *f(n)* and *g(n)*, we need a **rough measure** that characterizes **how fast each function grows.**
- *Hint:* use *rate of growth*
- Compare functions in the limit, that is, **asymptotically!**

  (i.e., for large values of *n*)

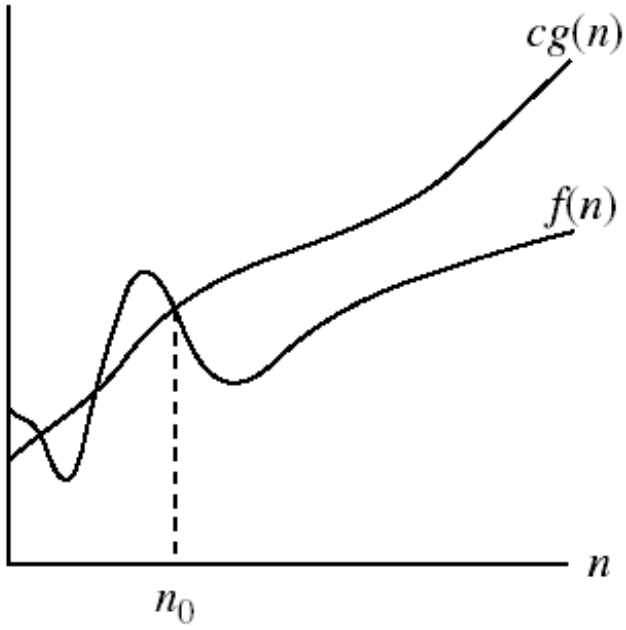
Asymptotic Notation

- O notation :Big-O is the formal method of expressing the upper bound of an algorithm's running time.
- It's a measure of the longest amount of time it could possibly take for the algorithm to complete.
- Formally, for non-negative functions, *f(n)* and *g(n)*, if there exists an integer n_0 and a constant $c > 0$ such that for all integers $n > n_0$, $f(n) \leq cg(n)$, then *f(n)* is Big O of *g(n).*

Asymptotic notations

- *O-notation*

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}.$$



$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

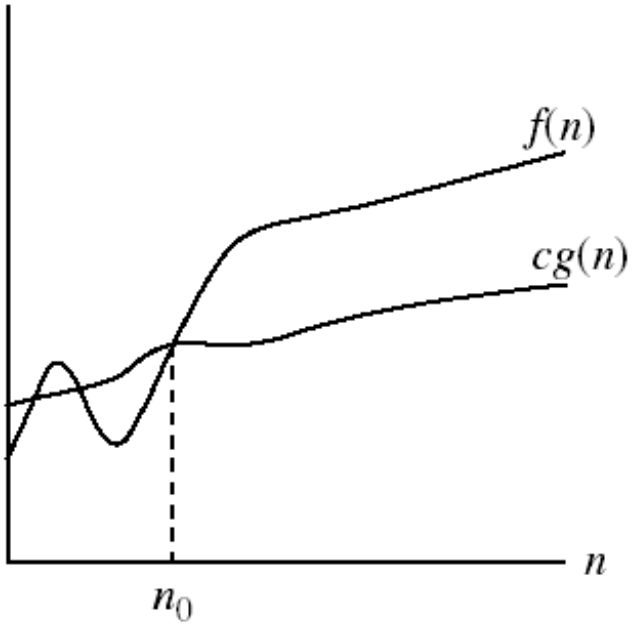
Asymptotic Notation

- ***Big-Omega Notation*** Ω
- This is almost the same definition as Big Oh, except that "$f(n) \geq cg(n)$"
- This makes $g(n)$ a lower bound function, instead of an upper bound function.
- It describes the **best that can happen** for a given data size.

- For non-negative functions, $f(n)$ and $g(n)$, if there exists an integer n_0 and a constant $c > 0$ such that for all integers $n > n_0$, $f(n) \geq cg(n)$, then $f(n)$ is omega of $g(n)$. This is denoted as "$f(n) = \Omega(g(n))$".

Asymptotic notations (cont.)

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\} .$$



Ω(g(n)) is the set of functions with larger or same order of growth as g(n)

$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

Asymptotic Notation

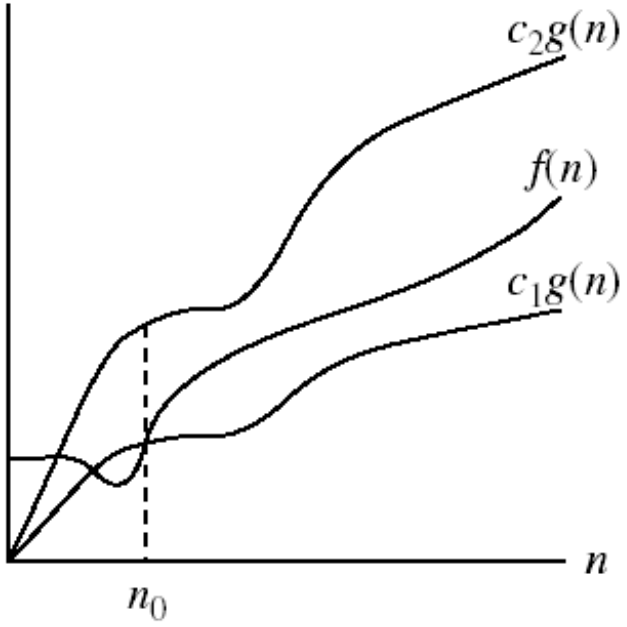
- **Theta Notation** Θ
- Theta Notation For non-negative functions, *f(n)* and *g(n)*, *f(n)* is theta of *g(n)* if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. This is denoted as "$f(n) = \Theta(g(n))$".

  This is basically saying that the function, *f(n)* is bounded both from the top and bottom by the same function, *g(n)*.

Asymptotic notations (cont.)

❖ Θ-notation

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\} .$$



Θ(g(n)) is the set of functions with the same order of growth as g(n)

$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

Asymptotic notations (cont.)

- **Little-O Notation**
- For non-negative functions, *f(n)* and *g(n)*, *f(n)* is little o of *g(n)* if and only if *f(n) = O(g(n))*, but *f(n) ≠ Θ(g(n))*. This is denoted as "*f(n) = o(g(n))*".
- This represents a loose bounding version of Big O. *g(n)* bounds from the top, but it does not bound the bottom.

- **Little Omega Notation**
- For non-negative functions, *f(n)* and *g(n)*, *f(n)* is little omega of *g(n)* if and only if *f(n) = Ω(g(n))*, but *f(n) ≠ Θ(g(n))*. This is denoted as "*f(n) = ω(g(n))*".

- It bounds from the bottom, but not from the top.

Example

- Associate a "cost" with each statement.
- Find the "total cost“ by finding the total number of times each statement is executed.

Algorithm 1

	Cost
arr[0] = 0;	c_1
arr[1] = 0;	c_1
arr[2] = 0;	c_1
...	...
arr[N-1] = 0;	c_1

$c_1+c_1+...+c_1 = c_1 \times N$

Algorithm 2

	Cost
for(i=0; i<N; i++)	c_2
arr[i] = 0;	c_1

$(N+1) \times c_2 + N \times c_1 = (c_2 + c_1) \times N + c_2$

Another Example

❖ ***Algorithm 3*** *Cost*

```
sum = 0;                                  c1
for(i=0; i<N; i++)                        c2
   for(j=0; j<N; j++)                     c2
      sum += arr[i][j];                   c3
```

$c_1 + c_2 \times (N+1) + c_2 \times N \times (N+1) + c_3 \times N^2$

Rate of Growth

- Consider the example of buying *elephants* and *goldfish:*

  Cost: cost_of_elephants + cost_of_goldfish

  Cost ~ cost_of_elephants (approximation)

- The low order terms in a function are relatively insignificant for **large** *n*

$$n^4 + 100n^2 + 10n + 50 \quad \sim \quad n^4$$

i.e., we say that $n^4 + 100n^2 + 10n + 50$ and n^4 have the same **rate of growth**

- FURTHER DISCUSSION

NEXT WEEK ……