Data Structures and Algorithms
EE2008/IM1001

COPYRIGHT STATEMENT

All course materials, including but not limited to, lecture slides, handout and recordings, are for your own educational purposes only. All the contents of the materials are protected by copyright, trademark or other forms of proprietary rights.

All rights, title and interest in the materials are owned by, licensed to or controlled by the University, unless otherwise expressly stated. The materials shall not be uploaded, reproduced, distributed, republished or transmitted in any form or by any means, in whole or in part, without written approval from the University.

You are also not allowed to take any photograph, film, audio record or other means of capturing images or voice of any contents during lecture(s) and/or tutorial(s) and reproduce, distribute and/or transmit any form or by any means, in whole or in part, without the written permission from the University.

Appropriate action(s) will be taken against you including but not limited to disciplinary proceeding and/or legal action if you are found to have committed any of the above or infringed the University's copyright.

Textbooks:

- Anany Levitin, *Introduction to The Design and Analysis of Algorithms*, 3rd ed., Pearson, 2012

- Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein, *Introduction to Algorithms*, 3rd Ed, The MIT Press, 2009 (QA76.6 I5858)

References:

- GB Huang and JM Ng (eds), *Data Structures and Algorithms*, Pearson, 2007

- Richard Johnsonbaugh and Marcus Schaefer, *Algorithms*, Prentice Hall, 2004

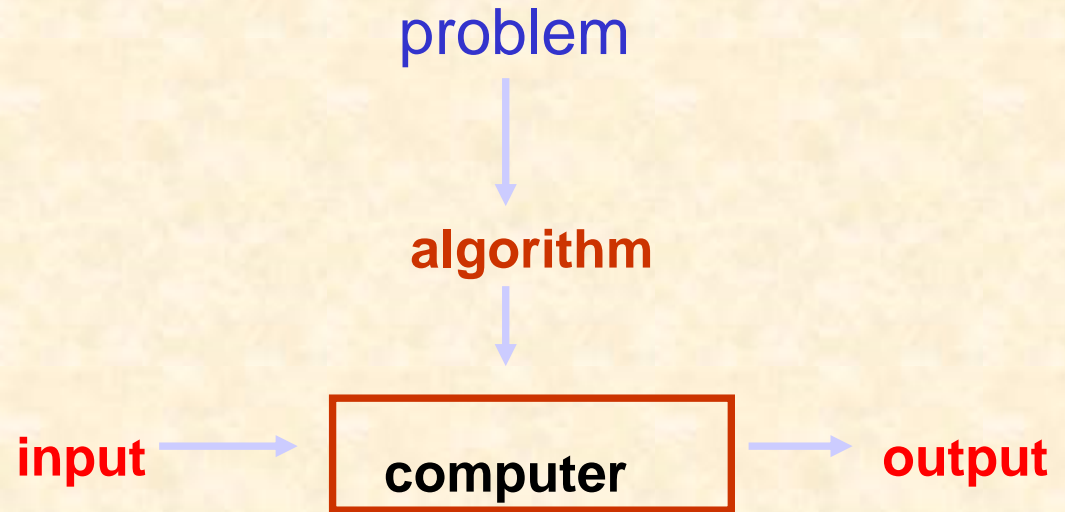# What are Algorithms & Data Structures??

What is the use of Algorithms ?

What is the use of data structures ?

What is the inter-relationship between algorithms & data structures ?

What is the use of Algorithms ???
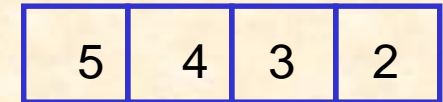
For solving a problem

problem

↓

algorithm

↓

input → | computer | → output

Algorithm

- **a sequence of instructions for solving a problem**

What is the use of Data Structure ??

Data Structure

ways of organizing data to promote efficient processing

Array

| 5 | 4 | 3 | 2 |

Linked List

| 5 | | → | 4 | | → | 3 | | → | 2 | |
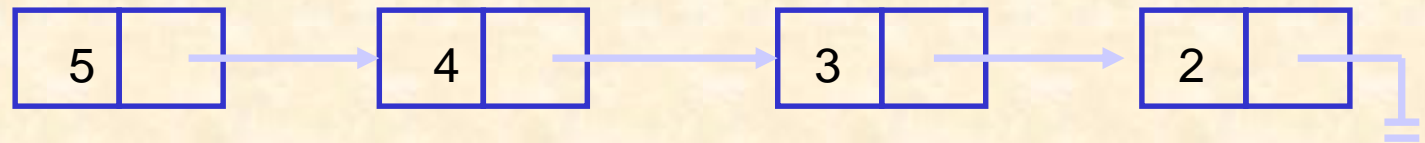
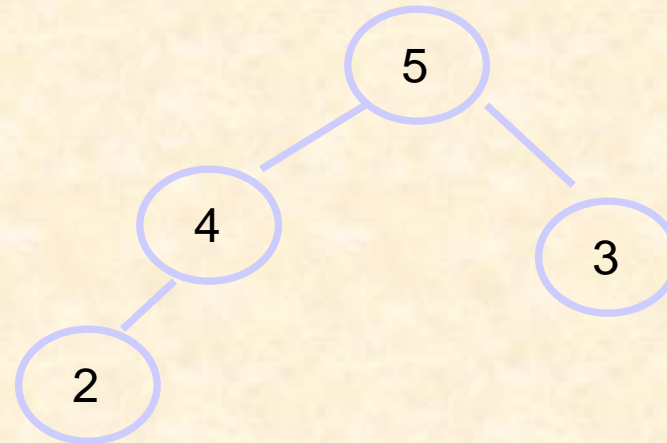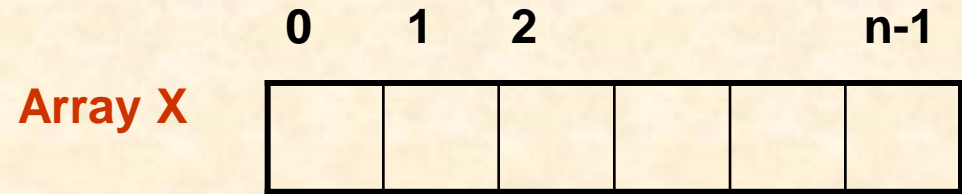**Max Heap**

What is the Relationship b/w Algorithms & Data Structure

Choice of Data Structures will affect efficiency of algorithm

- Example
  - **Input : a set of n numbers**
  - **Output: the kth element of the input list**

Array X

$$0 \quad 1 \quad 2 \qquad\qquad n\text{-}1$$

kth element = X(k-1)

Need 1 operation

$$1 \qquad 2 \qquad 3 \qquad k \qquad n$$

Linked List
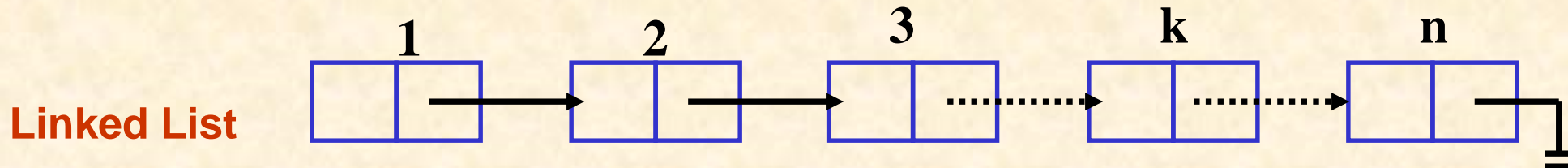
Need k-1 operations

Given a problem: choose appropriate data structure & algorithm to solve problem efficiently

Role of Algorithms/Data Structures in Modern World

- Algorithms are needed to solve real world problems Outline Principles of Algorithm Analysis Slide 61 Asy...

    - Internet Communication Slide 14

        - Because the Internet topology and network load is constantly changing, network routes must be discovered dynamically.

    - Network Security Slide 15

        - Prevent eavesdropping, modification, insertion or deletion of messages transmitted across the network

    - VLSI Design Automation . . . VLSI DESIGN Role of Algorithms/Data Structures in Modern...
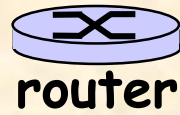
router

workstation

server
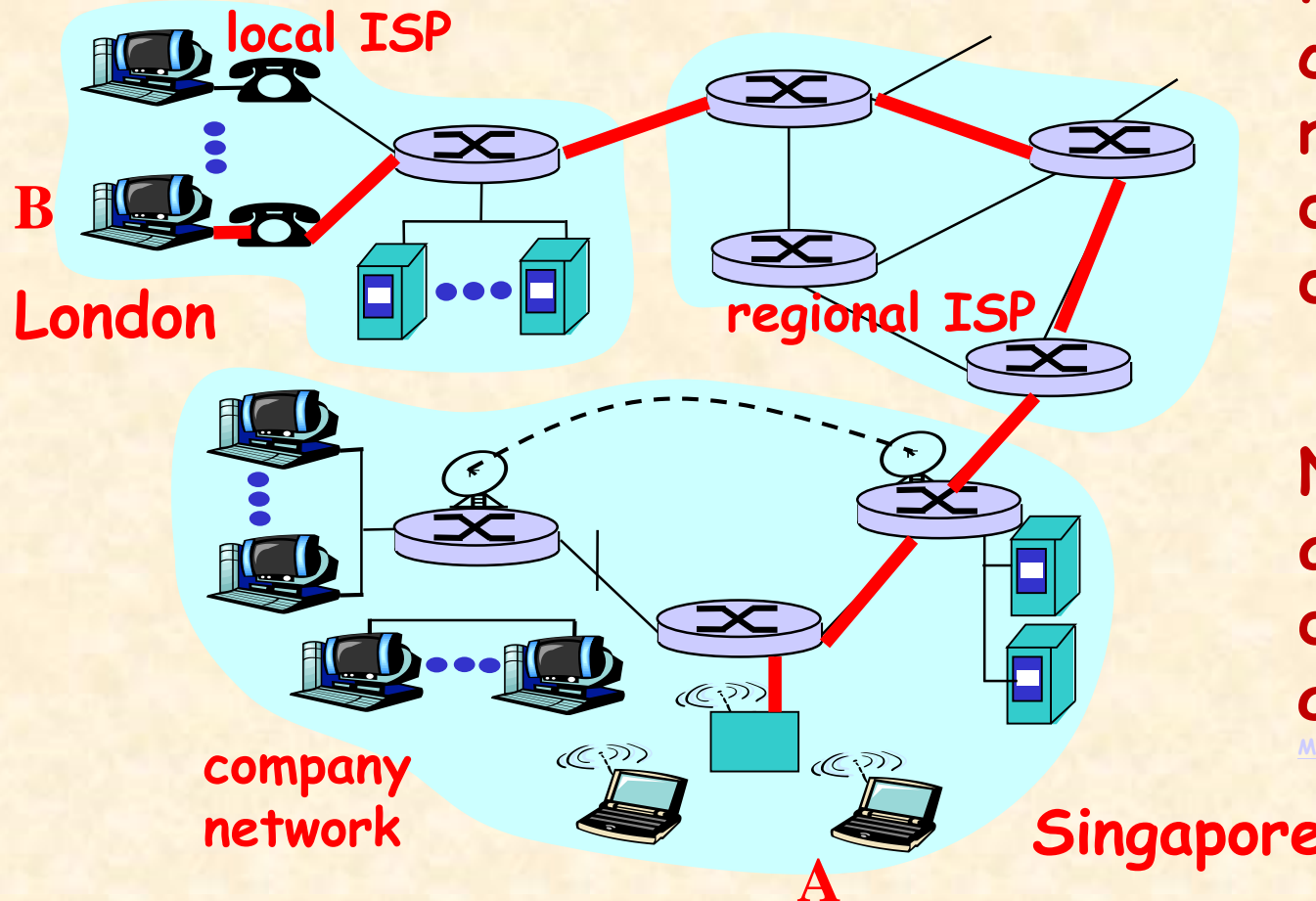
mobile

local ISP

B

London

regional ISP

company
network

Singapore

A

Because the Internet topology and network load is constantly changing, network routes must be discovered dynamically

Need efficient algorithms to discover new routes quickly

Role of Algorithms/Data Structures in Modern World

Network Security
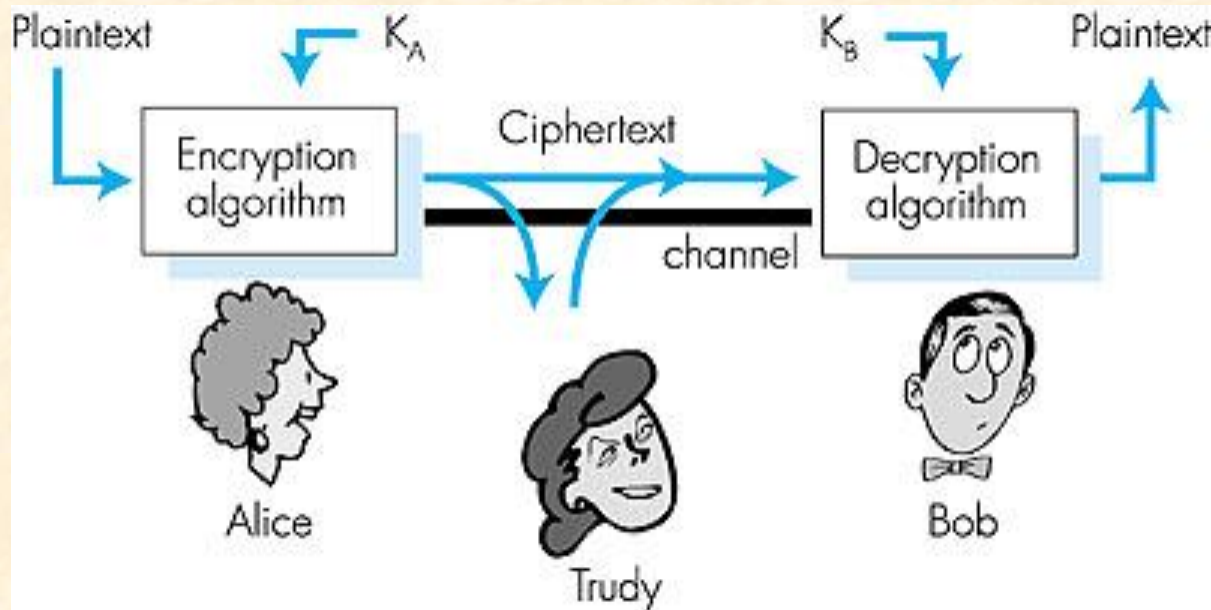
Encryption is the conversion of data into a form, called a Ciphertext, that cannot be easily understood by unauthorized people.

Decryption is the process of converting encrypted data back into its original form, so it can be understood.

Need algorithms to encrypt and decrypt transmitted data

Role of Algorithms/Data Structures in Modern World

VLSI DESIGN

- Devices in VLSI Circuits
  - Transistors
  - Logic gates and cells
  - Function blocks

- How to interconnect all the devices
  - Efficiently
  - Low power
  - Low cost

- **Need intelligent algorithms**
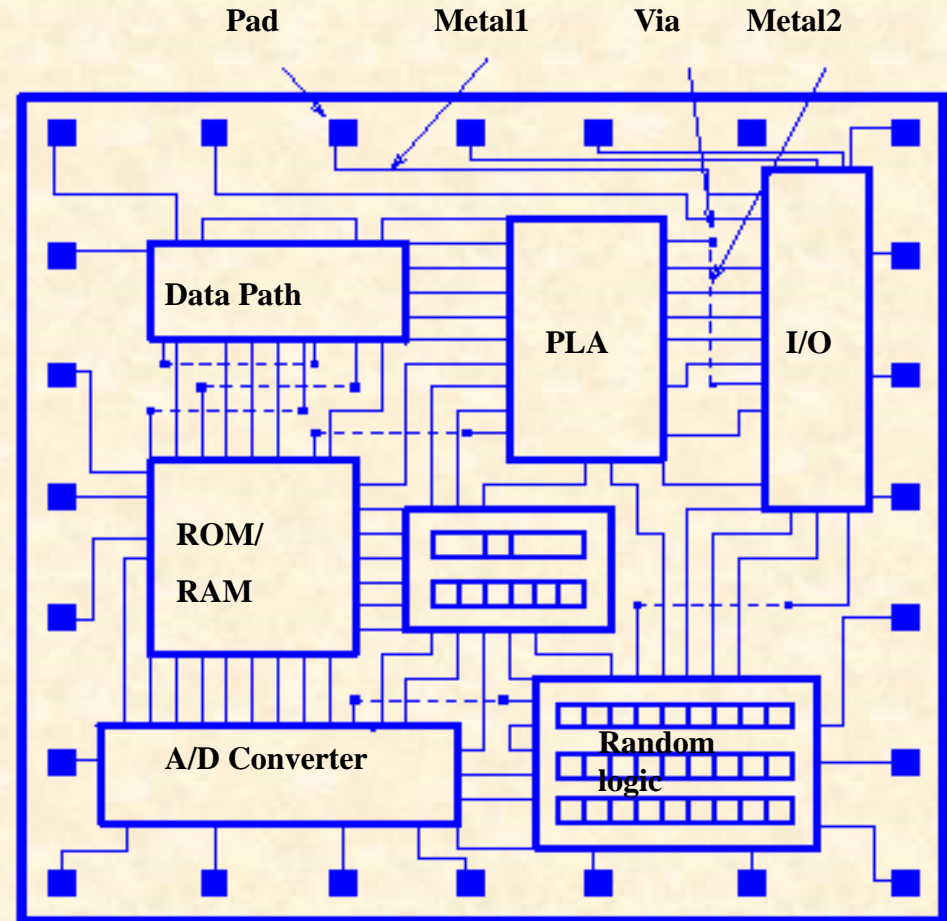
Outline Principles of Algorithm Analysis Slide 61   Asymptotics.pptx

- Overview of algorithms and data structures
- Basic mathematics for analysis of algorithms
- Asymptotic notation and Analysis of algorithms
  - This is an extremely useful mathematical technique because it simplifies greatly the analysis of algorithms
- Basic Recurrence Recursion: An Example Outline   Principles of Algorithm ...
  - Recursion is useful for problems that can be represented by a simpler version of the same problem. Notes_Part2_Jan09.pptx

- Basic Data Structures

Introduction

Overview

- What is an Algorithm?
  - A clearly specified set of instructions to be followed to solve a problem
    - takes a set of values as input and
    - produces a value or a set of values, as output

- Algorithms operate on data that are stored in a well defined structure called *data structure*, e.g. array, linked list, etc.
  - Data structure: a way of storing data so that the data can be used efficiently
    - a carefully chosen data structure will allow a more efficient algorithm to be used

- *Program = Algorithm + Data Structure*

Typical properties of Algorithms

- Input: The algorithm receives *input*.
- Output: produce *output*.
- Precision: steps are precisely stated.
- Determinism: results of each step are unique and are determined only by the inputs and results of the preceding steps.
- Finiteness: it **terminates**.
- Correctness: The output produced is correct.
- Generality: apply to a set of inputs.

Some Well-known Computational Problems

- Sorting

- Searching

- Graph Algorithms
  - Shortest paths in a graph
  - Minimum spanning tree

Sorting

- Given a sequence of numbers

- Rearrange the numbers in increasing order or decreasing order

   Example: 11, 7, 14, 1, 5, 9, 10

      Sort in increasing order

         11, 7, 14, 1, 5, 9, 10 $\rightarrow$ 1, 5, 7, 9, 10, 11, 14

An application: Sorting a deck of cards

Searching

- ## What is Searching?
  - retrieving information from a large amount of previously stored information

- ## What are the applications of searching?
  - **Banking**
    - keep track of all customers' account balances and to search through them to check for various types of transactions

  - **Search engine**: such as Google
    - need to look for relevant pages on the Web containing a given keyword
    - how does Google find the documents matching your query so fast?

Shortest Paths : An Example
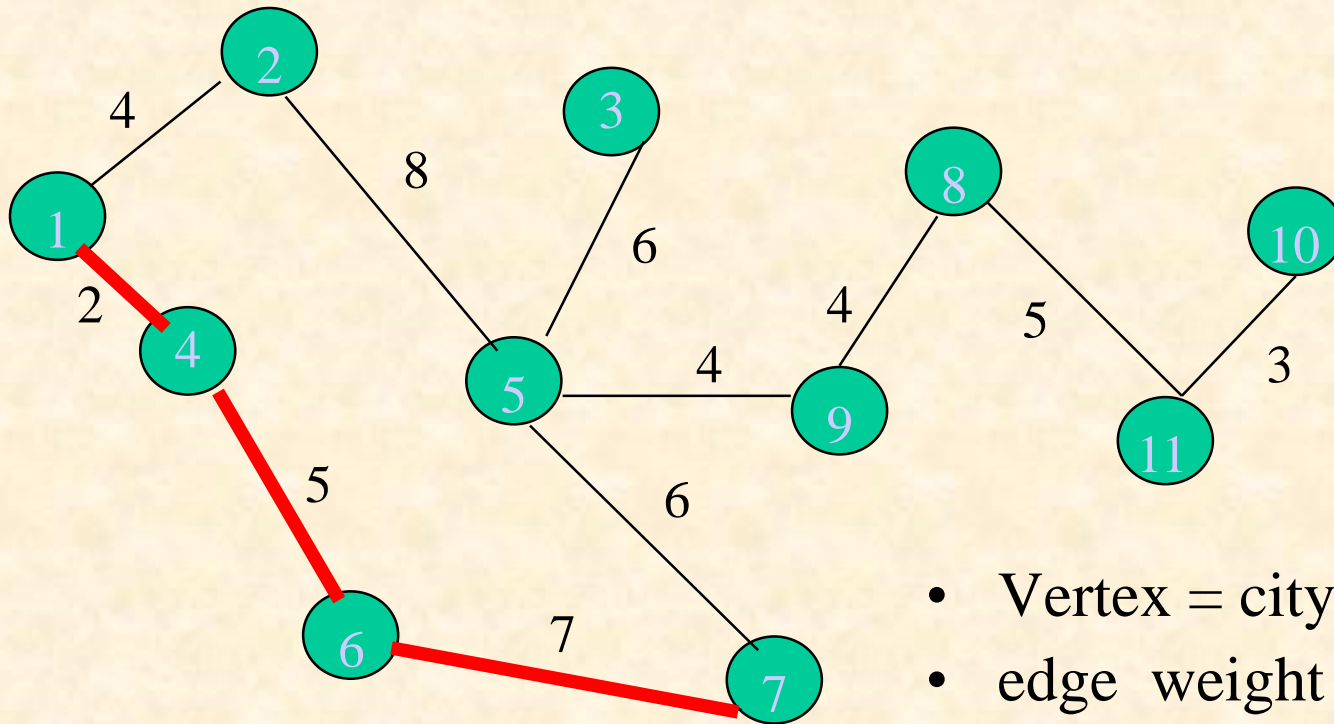
- Vertex = city,
- edge weight = driving distance/time.

What is the shortest path from 1 to 7?

Minimum Spanning Tree
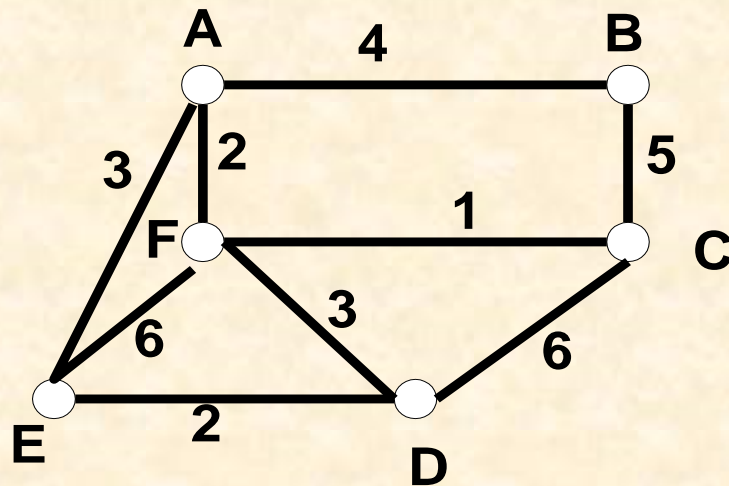
Each node represents a city

Weight of each edge: cost of building a road connecting two cities

Problem: to build enough roads so that each pair of cities will be connected and to use the lowest cost possible

Cost = 18

Cost = 12
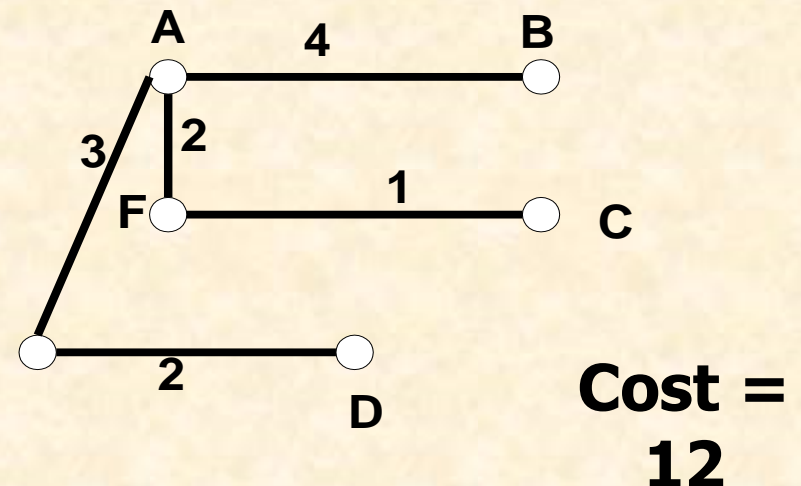
Sorting

- An Example: Sort a set of elements into increasing order

  11, 7, 14, 1, 5, 9, 10 $\rightarrow$ 1, 5, 7, 9, 10, 11, 14

Insertion sort

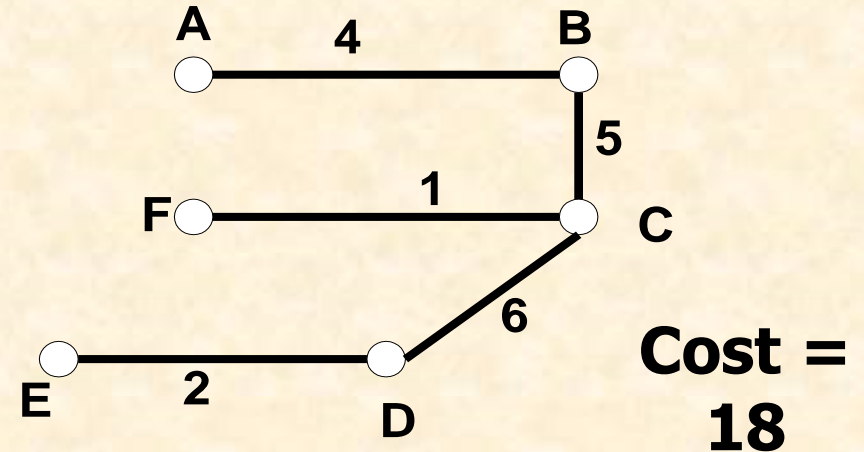- Examine the number from left to right one by one, and insert the number into an appropriate place in an already sorted sequence

- Insertion of a number is done by comparing it with the numbers in the sorted sequence

- The comparison stops when the correct position is determined

Sorted sequence	Unsorted sequence
11	7, 14, 1, 5, 9, 10
7, 11	14, 1, 5, 9, 10
7, 11, 14	1, 5, 9, 10
1, 7, 11, 14	5, 9, 10
1, 5, 7, 11, 14	9, 10
1, 5, 7, 9, 11, 14	10
1, 5, 7, 9, 10, 11, 14	

Quick Sort: 10, 5, 1, 17, 14, 8, 7, 26, 21, 3

- Use the first element to divide the numbers into 3 subsets: (smaller than 10) (10) (greater than 10)
  - (5, 1, 8, 7, 3) (10) (17, 14, 26, 21)
- Next, sort the left and right subsets in the similar way
  - (1, 3) (5) (8, 7) and (14) (17) (26, 21)
- After sorting (8, 7) and (26, 21), and combining the sorted subsets, we have
  - (1, 3, 5, 7, 8) and (14, 17, 21, 26)
- Finally, combining the sorted subsets, we have
  - 1, 3, 5, 7, 8, 10, 14, 17, 21, 26

Quick Sort: 10, 5, 1, 17, 14, 8, 7, 26, 21, 3

(1,3,5,7,8, 10,14,17,21,26)

↑

(5, 1, 8, 7, 3) (10) (17, 14, 26, 21)

↑↓   ↓↑

(1,3,5,7,8)   (14,17,21,26)

(1, 3) (5) (8, 7)   (14) (17) (26, 21)

(1,3)   ↑↓   ↓↑ (7,8)   ↓↑ (21,26)

(1)(3)   (7)(8)   (21)(26)
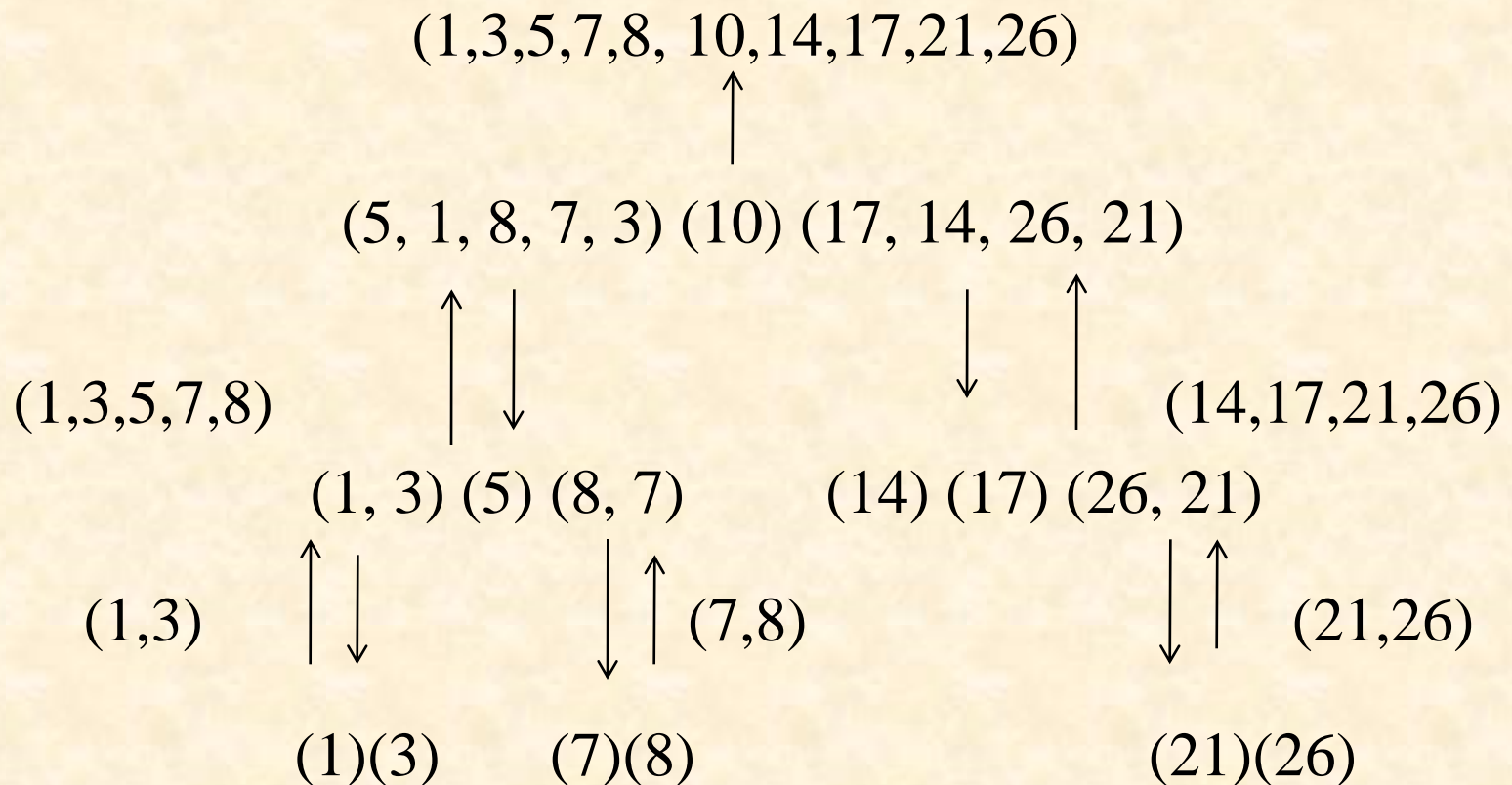
Comparing the Performance of Insertion Sort and Quick Sort (time in sec)

n	Insertion sort	Quick sort
10	.000001	.000002
100	.000106	.000025
1,000	.011240	.000365
10,000	1.047	.004612
100,000	110.492	.058481
1,000,000	NA	.6842

This example shows that a same problem can be solved by more than one algorithms with different level of efficiencies

When n is large, insertion sort will take a longer time to execute than Quick Sort even when a much faster computer is used.

Basic Mathematics for Algorithms

Sets

- A set is a collection of objects, called its members or elements
- If an object x is a member of S, **we write $x \in S$**
- If x is not a member of S, **we write $x \notin S$**
- If S contains elements a, b, c, **we write $S = \{ a, b, c \}$**
  - is $a \in S$?
  - is b $\in S$?
  - is $c \in S$?
  - is $d \in S$?

- $\varnothing$: denotes an empty set (a set with no members)

Sets (contd)

- If all the elements of a set A are contained in a set B, **then we write A $\subseteq$ B (A is a subset of B)**

- If A $\subseteq$ B but A $\neq$ B

  then we write A$\subset$ B (A is a proper subset of B)

Set Operations

- The intersection of sets A and B
    - $A \cap B = \{x : x \in A \text{ and } x \in B\}$
    - Eg; $A = \{1,2,3\}$, $B = \{2,3,5,6\}$
        - $A \cap B = \{2, 3\}$

- The union of sets A and B
    - $A \cup B = \{x : x \in A \text{ or } x \in B\}$
    - Eg: $A = \{1,2,3\}$, $B = \{2,3,5,6\}$
        - $A \cup B = \{1,2,3,5,6\}$

- The difference between 2 sets A and B

    $A - B = \{x : x \in A \text{ and } x \notin B\}$

    Eg: $A = \{1,2,3\}$, $B = \{2,3,5,6\}$, **then $A - B = \{1\}$**

Definitions and Notations

- Basic notation:
  - Z : set of integers $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$
  - N: set of natural numbers $\{1, 2, \ldots\}$
  - R: set of real numbers Numbers   Definitions and Notations
  - floor: $\lfloor x \rfloor$ = largest integer less than or equal to x. $\lfloor 4.9 \rfloor = 4$
  - ceiling: $\lceil x \rceil$ = smallest integer greater than or equal to x. $\lceil 7.1 \rceil = 8$

- Polynomials: Big-Oh Rules
  - A polynomial of degree n is a function of the form
  $$p(x) = c_n x^n + c_{n-1} x^{n-1} + \ldots + c_1 x + c_0$$
  with $c_n \neq 0$. The numbers c_i are called coefficients.
  - Example, a polynomial of degree 5.
  $$p(x) = 3x^5 - 12x^3 + 9x + 4$$

Intervals

- If a and b are numbers, $[a, b]$ is called a ***closed interval*** and is defined to be the set $\{x \mid a \le x \le b\}$

  - if we are restricted to integers, $[a, b]$ is the set:

    $\{x \mid x$ is an integer and $a \le x \le b\}$

    - E.g. $[2,6] = \{2,3,4,5,6\}$

  - if we are restricted to real numbers, $[a, b]$ is the set:

    $\{x \mid x$ is a real number and $a \le x \le b\}$

    - e.g. $2 \in [2,6]$, $\pi \in [2,6]$

- If a and b are number, (a, b) is called an ***open interval*** and is defined to be the set $\{x \mid a < x < b\}$

- $[a, b)$ and $(a, b]$ are called ***half-open interval***.

  - $[a, b) : \{x \mid a \le x < b\}$; $(a, b] : \{x \mid a < x \le b\}$

Functions

- A function from a set X to a set Y is a relationship between the elements of X and the elements of Y

- Property of the relationship : each element of X is related to a unique element of Y

- The notation f : X $\longrightarrow$ Y means that f is a function from X to Y

$f(x_1) = y_3$

$f(x_2) = y_1$

$f(x_3) = y_4$
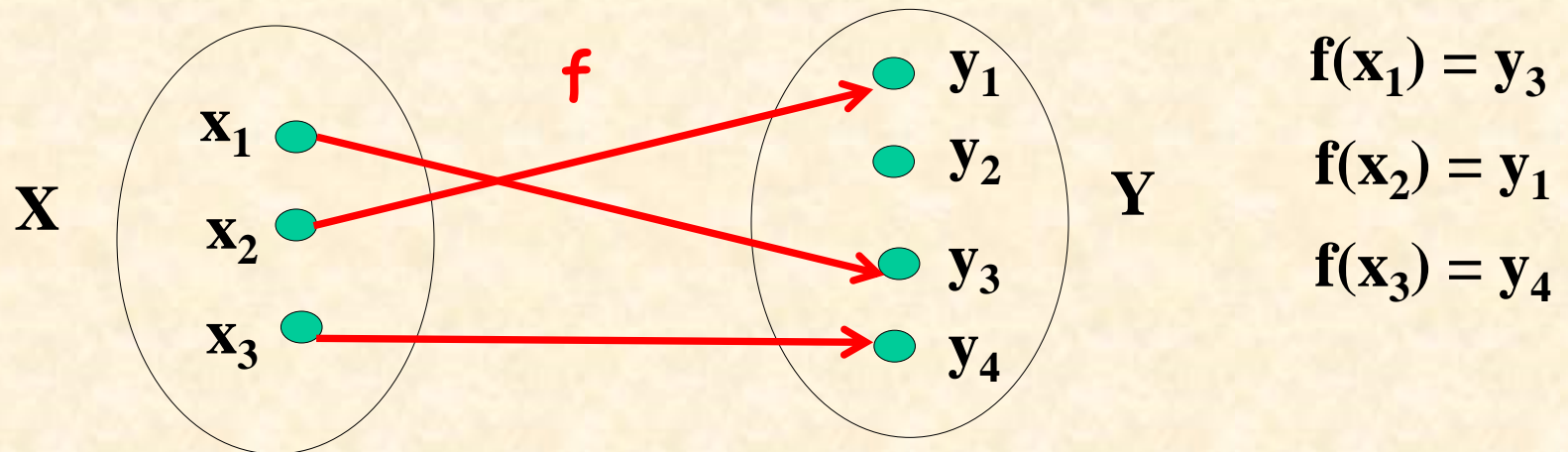
- Sequences:
  - A *finite sequence* a is a function from the set $\{0,1, .., n\}$ to a set X.
    - The sequence is typically denoted as $a_0, a_1, \ldots$, the subscript i in a_i is the index of the sequence. Sequences_Slide 50
  - An *infinite sequence* a is a function from the set $\{0, 1, \ldots\}$ to a set X. The sequence is denoted $a_0, a_1, \ldots$, or
    - E.g. $a_i = 3i + 1$, $i \geq 0$, is a infinite sequence,
    - 4, 7, 10, …
  - a is an **increasing sequence** if $\forall i, a_i < a_{i+1}$

  $$\{a_i\}_{i=0}^{\infty}$$

  - a is a **decreasing sequence** if $\forall i, a_i > a_{i+1}$
  - a is a **non-increasing sequence** if $\forall i, a_i \geq a_{i+1}$, e.g. 100,90,74,74,56
  - a is a **non-decreasing sequence** if $\forall i, a_i \leq a_{i+1}$

- Strings:
  - A string or word is a finite sequence $t_0, t_1, \ldots, t_n$, where $t_i \in X$, and is written as $t_0 t_1 \ldots t_n$.
    - E.g. $X=\{a,b,c\}$, $t_0=b$, $t_1=a$, $t_2=a$, $t_3=c$, the string is written as baac.

Sequences

- A finite sequence
  - $f : \{0,1,2,\ldots,n\} \; --> X$
  - $f(i) = a_i \in X$
  - $a_0, \; a_1, \; a_2, \; \ldots, \; a_n$
- An infinite sequence
  - $f : \{0,1,2,\ldots,\} \rightarrow X$
  - $a_0, \; a_1, \; a_2, \; \ldots$
- 1,3,5,7, … (an increasing sequence)
- 100, 98, 76, 55, … (a decreasing sequence)
- 100, 98, 76, 76, 55, 54, … (a non-increasing sequence)
- 1, 3, 5, 5, 7, 8, … (a non-decreasing sequence)
- If X is a set of alphabets, $f:\{0,1,2, \ldots,n\} \rightarrow X$ forms a string of alphabets or words
  - Eg $X = \{r,s,t\}$, $f(0) = r$, $f(1) = s$, $f(2)=r$, $f(3)=t$, the string of alphabets or word formed is rsrt

Boolean (Logic) Variables

- Boolean variables have only two possible values: T, F
- Boolean operators: *or* ($\vee$), *and* ($\wedge$), *not* ($\bar{}$)
- Precedence
  - *not* ($\bar{}$) (highest precedence)
  - *and* ($\wedge$)
  - *or* ($\vee$) (lowest precedence)
- Example: $p \vee q \wedge \bar{r}$

  is interpreted as *p or (q and (not r))*

Suppose p = T, q = T and r = T

Then $\bar{r} =$ F

(q and (not r)) = F

p or (q and (not r)) = T

p	q	$p \vee q$	$p \wedge q$
T	T	T	T
T	F	T	F
F	T	T	F
F	F	F	F

- Factorials:

  $n! = n \times (n-1) \times \ldots \times 3 \times 2 \times 1$

  Example: $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

- Binomial Coefficient $\binom{n}{k}$

  – Provides the number of **combinations** of selecting k elements from a set with n elements

  – For $n \geq k \geq 0$, the number of k-element subsets of an n-element set is given by

  – Combinations_Slide 52

  $$\binom{n}{k} = \frac{n!}{(n-k)!\,k!}$$

  – Example:

  $$\binom{5}{2} = \frac{5!}{3!2!} = \frac{5 \times 4 \times 3 \times 2 \times 1}{(3 \times 2 \times 1) \times (2 \times 1)} = 10$$

Combinations

- $\binom{n}{k}$: the number of ways of selecting k elements from a set of n elements

$$\binom{n}{k} = \frac{n!}{(n-k)!k!}$$

- Example: Given a set of numbers {1,2,3,4}, how many subsets of two numbers can be formed?
  - {1,2}, {1,3}, {1,4}
  - {2,3}, {2,4}
  - {3,4}

$$\binom{4}{2} = \frac{4!}{2!2!} = \frac{4 \times 3 \times 2 \times 1}{(2 \times 1)(2 \times 1)} = 6$$

- Logarithms:
  - Recall: if $b^x = n$, $\log_b n = x$ **Definition of logarithm**
  - Laws of logarithms: Suppose that $b > 0$, and $b \neq 1$, then

$$b^{\log_b x} = x$$

$$\log_b(xy) = \log_b x + \log_b y$$

$$\log_b(\frac{x}{y}) = \log_b x - \log_b y$$

$$\log_b x^y = y \log_b x$$

$$if\ a > 0\ and\ a \neq 1, \log_a x = \frac{\log_b x}{\log_b a} \quad \textbf{Change of base}$$

$$if\ b > 1\ and\ x > y > 0, \log_b x > \log_b y$$

  - Note: $\log_2$ is usually written as lg

Examples

1. Let $b=2$ and $x=8$, then $\log_2 x = 3$

   Now $b^{\log_b x} = 2^3 = 8 = x$

2. $\log_{10} 32 = 1.505, \quad \log_{10} 2 = 0.301$

$$\log_2 32 = \frac{\log_{10} 32}{\log_{10} 2} = \frac{1.505}{0.301} = 5$$

   Note: $2^5 = 32, \quad \log_2 32 = 5$

Basic Probability

- S = sample space
  - (a set of all possible outcomes of an experiment)
  - Eg: consider an experiment of flipping a coin
    - Two possible outcomes : head or tail
    - The chance of getting a head is 0.5
      - P(getting a head) = 0.5
    - The chance of getting a tail is 0.5
      - P(getting a tail) = 0.5
    - The P(getting a head or a tail) = 0.5 + 0.5 = 1

- each subset A of S is called an event
  - $0 \leq P(A) \leq 1$
  - P(S) = 1

Random Variables & Expectation

- Random variables : variables whose values depend on the outcome of some experiment

- Example : X is a random variable with possible values {1,2,3}. Suppose each of the possible values if equally likely to occur

  – Then the expected (average) value of $X = \dfrac{1+2+3}{3}$

  – $E(X) = (1)\dfrac{1}{3} + (2)\dfrac{1}{3} + (3)\dfrac{1}{3}$

- In general, given a random variable X, the expected (average) value of X is

$$E(X) = \sum_r r P(X=r)$$

Summation

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n$$

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + 3^2 + \cdots + n^2$$

$$\sum_{i=1}^{n} a = \underbrace{a + a + a + \cdots a}_{n\ times} = na$$

$$\sum_{i=p}^{q} a = \underbrace{a + a + a + \cdots a}_{q-p+1\ times} = (q - p + 1)a$$

Sum Manipulation Rules

$$\sum_{i=p}^{u} ca_i = c\sum_{i=p}^{u} a_i$$

$$\sum_{i=1}^{10} 3i = 3\sum_{i=1}^{10} i$$

$$\sum_{i=p}^{u} (a_i \pm b_i) = \sum_{i=p}^{u} a_i \pm \sum_{i=p}^{u} b_i$$

$$\sum_{k=1}^{100} 3^k + k^2 = \sum_{k=1}^{100} 3^k + \sum_{k=1}^{100} k^2$$

$$\sum_{i=p}^{u} a_i = \sum_{i=p}^{r} a_i + \sum_{i=r+1}^{u} a_i \quad \textbf{where } p \le r \le u$$

$$\sum_{i=1}^{1000} i = \sum_{i=1}^{100} i + \sum_{i=101}^{1000} i$$

Mathematical Induction

- Mathematical induction can be used to prove a statement is true for all natural numbers $n \geq n_o$ if
  - (Basis of induction) The statement is true for $n = n_o$; and
  - (Induction step) Assuming that the statement is true for $n = k, k \geq n_o$, prove that the statement is true for $n = k + 1$.
  - Let p(n) : Statement to be proven true for $n \geq n_o$

- Example: Prove that

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n \quad = \frac{n(n+1)}{2} \qquad \text{for all } n \geq 1$$

- Basis step: Show that the equation is true for $n = 1$.

$$LHS = \sum_{i=1}^{1} i = 1 \qquad RHS = \frac{1(1+1)}{2} = 1$$

- Inductive step: assume that equation is true for n, and prove that it is true for $n + 1$.

$$\sum_{i=1}^{n+1} i = \sum_{i=1}^{n} i + (n+1) = \frac{n(n+1)}{2} + (n+1)$$

$$\Rightarrow \sum_{i=1}^{n+1} i = \frac{n(n+1) + 2(n+1)}{2}$$

$$\Rightarrow \sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2}$$

Induction: Example

- P(n) : $\displaystyle\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ for all n ≥ 1

- Inductive step: Assume P(k) is true

$$\sum_{i=1}^{k} i = \frac{k(k+1)}{2}$$

- Need to prove P(k+1) is true

$$\sum_{i=1}^{k+1} i = \frac{(k+1)(k+2)}{2}$$

- We have $\displaystyle\sum_{i=1}^{k} i = \frac{k(k+1)}{2}$

- Need to show $\displaystyle\sum_{i=1}^{k+1} i = \frac{(k+1)(k+2)}{2}$

- $\displaystyle\sum_{i=1}^{k+1} i = \sum_{i=1}^{k} i + (k+1) = \frac{k(k+1)}{2} + (k+1)$

$$= \frac{k(k+1) + 2(k+1)}{2}$$

$$= \frac{(k+1)(k+2)}{2}$$

- Example: Prove that

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + ... + n^2 = \frac{n(n+1)(2n+1)}{6} \quad \text{for all } n \geq 1$$

  - Basis step: Show that the equation is true for $n = 1$.

$$LHS = \sum_{i=1}^{1} i^2 = 1^2 = 1 \qquad RHS = \frac{1(1+1)(2+1)}{6} = 1$$

  - Inductive step: assume that equation is true for n, and prove that it is true for $n + 1$. Induction Example 2

$$\sum_{i=1}^{n+1} i^2 = \sum_{i=1}^{n} i^2 + (n+1)^2 = \frac{n(n+1)(2n+1)}{6} + (n+1)^2$$

$$\Rightarrow \sum_{i=1}^{n+1} i^2 = \frac{(n+1)[n(2n+1) + 6(n+1)]}{6}$$

$$\Rightarrow \sum_{i=1}^{n+1} i^2 = \frac{(n+1)(2n^2 + 7n + 6)}{6}$$

$$\Rightarrow \sum_{i=1}^{n+1} i^2 = \frac{(n+1)(n+2)(2n+3)}{6}$$

$$\Rightarrow \sum_{i=1}^{n+1} i^2 = \frac{(n+1)[(n+1)+1][(2(n+1)+1]}{6}$$

Induction Example

- P(n) : $\displaystyle\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$

- Assume P(k) is true

$$\sum_{i=1}^{k} i^2 = \frac{k(k+1)(2k+1)}{6}$$

- Need to prove P(k+1) is true

$$\sum_{i=1}^{k+1} i^2 = \frac{(k+1)(k+2)(2k+3)}{6}$$

- We have
$$\sum_{i=1}^{k} i^2 = \frac{k(k+1)(2k+1)}{6}$$

- Need to show :
$$\sum_{i=1}^{k+1} i^2 = \frac{(k+1)(k+2)(2k+3)}{6}$$

$$\sum_{i=1}^{k+1} i^2 = \sum_{i=1}^{k} i^2 + (k+1)^2 = \frac{k(k+1)(2k+1)}{6} + (k+1)^2$$

$$= \frac{(k+1)[k(2k+1) + 6(k+1)]}{6}$$

$$= \frac{(k+1)[2k^2 + 7k + 6]}{6}$$

$$= \frac{(k+1)(k+2)(2k+3)}{6}$$

Arithmetic Series

- **arithmetic sequence (series)** is a sequence of numbers such that the difference between the consecutive terms is constant.
  - Eg 1, 3, 5, 7, 9, 11,… (common difference is 2)
- General form
  - 1st term = a
  - 2nd term = a + d
  - 3rd term = a + 2d
  - nth term = a +(n-1)d

Geometric Series

- a **geometric series** is a series with a constant ratio between successive terms
  - Eg: 2, 4, 8, 16, 32,… (common ratio = 2)
- General form
  - 1st term = a
  - 2nd term = ar
  - 3rd term = $ar^2 + 2d$
  - nth term = $ar^{(n-1)}$

– Arithmetic series

$$\sum_{k=1}^{n}\left(a+(k-1)d\right)=a+(a+d)+(a+2d)+\cdots+(a+(n-1)d)=\frac{n}{2}\left(2a+(n-1)d\right))$$

Special case :
$$\sum_{i=1}^{n}i=1+2+...+n=\frac{n(n+1)}{2}$$

– Geometric series

$$\sum_{k=0}^{n}ar^{k}=a+ar+...+ar^{n}=\frac{a(r^{n+1}-1)}{r-1}\qquad \mathbf{r>1}$$

–

Note $\quad\sum_{k=0}^{n}ar^{k}=\dfrac{a(1-r^{n+1})}{1-r}\qquad$ for $0<r<1$

– Telescoping series $\sum_{k=1}^{n}(a_{k}-a_{k-1})=a_{n}-a_{0}$
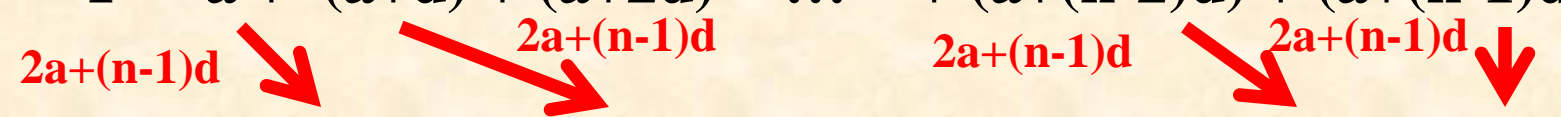
Arithmetic Series

$$\sum_{k=1}^{n}\left(a+(k-1)d\right) = a+(a+d)+(a+2d)+\cdots+(a+(n-1)d) = \frac{n}{2}\left(2a+(n-1)d)\right)$$

Let   **T = a + (a+d) + (a+2d) + … + + (a+(n-2)d) + (a+(n-1)d)**

2a+(n-1)d   **2a+(n-1)d**   **2a+(n-1)d**   **2a+(n-1)d**

Note that   **T = (a+(n-1)d) + (a+(n-2)d) + … + (a+2d) +(a+d) + a**

2T = n[2a+(n-1)d]

$$T = \frac{n}{2}[2a + (n-1)d]$$

Arithemetic Series: A Special Case

$$\sum_{k=1}^{n} \left(a + (k-1)d\right) = a + (a+d) + (a+2d) + \cdots + (a+(n-1)d) = \frac{n}{2}\left(2a + (n-1)d\right)$$

When $a = 1$ and $d = 1$, we have : $1 + 2 + 3 + \ldots + n$

$$= \frac{n}{2}[2 + (n-1)]$$

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

We have earlier proven that

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

A special case of an Arithmetic Series

Geometric Series : Example

$$\sum_{k=0}^{10} 2(3)^k = 2(3)^1 + 2(3)^2 + 2(3)^3 + \cdots + 2(3)^{10}$$

$$= \frac{2(3^{10+1} - 1)}{3 - 1}$$

$$\sum_{k=0}^{n} ar^k = \frac{a(r^{n+1} - 1)}{r - 1}$$

$$= \frac{2(3^{11} - 1)}{2}$$

$$= 3^{11} - 1$$

Induction Example

- P(n) : $$\sum_{i=0}^{n} ar^i = \frac{a(r^{n+1}-1)}{r-1}$$

- Basic Step: n=0

  - LHS $= \sum_{i=0}^{0} ar^i = a$

  - RHS $= \dfrac{a(r^{0+1}-1)}{r-1} = a$

Induction Example

- Assume P(k) is true $$\sum_{i=0}^{k} ar^i = \frac{a(r^{k+1}-1)}{r-1}$$

- Need to prove P(k+1) is true

$$\sum_{i=0}^{k+1} ar^i = \frac{a(r^{k+2}-1)}{r-1}$$

- We have $\displaystyle\sum_{i=0}^{k} ar^i = \frac{a(r^{k+1}-1)}{r-1}$

- Need to show : $\displaystyle\sum_{i=0}^{k+1} ar^i = \frac{a(r^{k+2}-1)}{r-1}$

$$\sum_{i=1}^{k+1} ar^i = \sum_{i=1}^{k} ar^i + ar^{k+1} = \frac{a(r^{k+1}-1)}{r-1} + ar^{k+1}$$

$$= \frac{a(r^{k+1}-1) + ar^{k+1}(r-1)}{r-1}$$

$$= \frac{a(r^{k+1}-1) + ar^{k+2} - ar^{k+1}}{r-1}$$

$$= \frac{a(r^{k+2}-1)}{r-1}$$

Geometric Series

$$Let \ P = \sum_{k=1}^{n} r^k = 1 + r + r^2 + r^3 + \dots + r^n$$

$$rP = \quad r + r^2 + r^3 + \dots + r^n + r^{n+1}$$

$$P - rP = 1 - r^{n+1}$$

$$P(1 - r) = 1 - r^{n+1}$$

$$P = \frac{(1 - r^{n+1})}{1 - r}$$

Telescoping Series

$$\sum_{k=1}^{n}\left(a_k - a_{k-1}\right) = (a_1 - a_0) + (a_2 - a_1) + (a_3 - a_2) + \ldots + (a_{n-1} - a_{n-2}) + (a_n - a_{n-1})$$

$$= a_n - a_0$$

Describing Algorithms : Using Pseudocode

Algorithm

      - a sequence of instructions for solving a   problem

Pseudocode

- High-level description of an algo.
- More structured than English prose
- Less detailed than a program
- 3 basic control constructs: sequence, selection and iteration
- Control constructs must be reflected clearly by some standard conventions (many options are available)

Example: find the largest element in an array

Algorithm *arrayMax*(A, n)
  Input array A of n integers
  Output largest element in A

  Max $= A[0]$
  for $i = 1$ **to** $n - 1$ **do**
    if $A[i] > Max$ **then**
      Max $= A[i]$
  return *Max*

0	1	2	3	4

A

Pseudocode Details

- Control flow

  if … **then** … [**else** …]

  while … **do** …

  repeat … **until** …

  for … **do** …

- Algorithm declaration

  Algorithm *name* (***arg***, ***arg,***…)

  Input …

  Output …

- Calling another algo.

  Algo_called (***arg***, ***arg,***…)

- Return value

  return *expression*

- Expressions

  =   Assignment $\mathbf{x\ =\ b}$

  == Equality testing   **if x = = b**

  n^2 Superscripts and other mathematical formatting allowed

Pseudocode: Selection

```
if (condition)
    action1;
```

```
if (condition)
    action1;
else
    action2;
```

```
if (b > x) // if b is larger than x, update x
    x = b
if (c > x) // if c is larger than x, update x
    x = c;
return x
```

```
if ( x ≥ 0) {
    x = x + 1;
    a = b + c;
}
```

Pseudocode: while loop

```
while (condition)
    action;
```

		1	2	3	4	5
s		-3	20	450		

- ✓ If *condition* is true, action is executed.
- ✓ The process is repeated until condition becomes **false**

```
// Find the max value in an array using a while loop
array_max1 (s) {
    large = s[1];
    i = 2;
    while (i ≤ s.last ) {
        if (s[i] > large) // larger value found
            large = s[i];
        i = i + 1;
    }
    return large;
}
```

Pseudocode: for loop

	1	2	3	4	5
	-3	20	450		

```
for (var = init to limit)
    action;
```

s

✓ **The variable *var* is first set to the value *init*. If *var* ≤ *limit*, action is executed and 1 is added to *var*.**

✓ **The process is repeated until *var* > *limit***

```
// Find the max value in an array using a for loop
array_max2 (s) {
    large = s[1];
    for (i = 2 to s.last ) {
        if (s[i] > large) // larger value found
            large = s[i];
    }
    return large;
}
```

Pseudocode: for loop

```
for (var = init downto limit)
    action;
```

✓ **Action is executed as long as *var ≥ limit*, and updating is performed by subtracting 1 from *var***

Write an algorithm that returns the *index* of the last occurrence of the value *key* in the array s[1], . . . , s[n]. If *key* is not in the array, the algorithm returns the value 0. Pseudocode: for loop

```
find_last_key (s, key) {
    for (i = s.last downto 1 ) {
        if (s[i] == key ) // key value found
            return i;
    }
    return 0;
}
```

✓ **== is used to test equal**
✓ **= is assignment**

	0	1	2	3	4
S	2	3	2	23	

Key = 2

Find index of last occurrence of key in array S

```
find_last_key (s, key) {
    for (i = s.last downto 1 ) {
        if (s[i] == key ) // key value found
            return i;
    }
    return 0;
}
```

Example 1

- Write an algorithm than compute the sum of n consecutive integers which starts from 1 (i.e. sum = 1 + 2 +3 + …. + n)

```
total(n)
    sum = 0;
    for (i = 1 to n ) {
        sum = sum + i;
    }
    return sum;
```

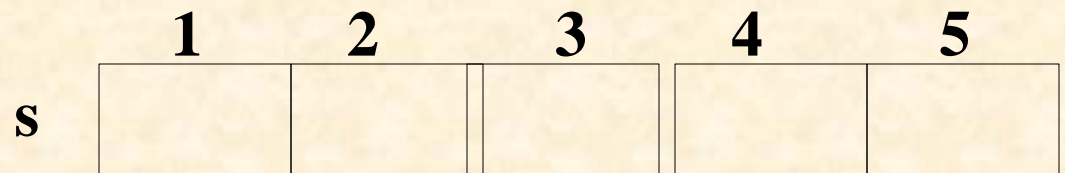| i	sum
1	1
2	1+2
3	1+2+3
n	1+2+3 + …+ n

Example 2

Write an algorithm to change a numeric grade to a pass/no pass grade
(pass grade $\geq 50\%$)

```
Grading(marks)
    if (marks ≥ 50){
        grade = "pass";
    else
        grade = "fail"
    }
    return grade;
```

Example 3

- Write an algorithm that returns the smallest value in the array s[1], s[2], …, s[n]. Use a for loop.

```
Array_min(s)
    min = s[1];
    for (i = 2 to n ) {
        if (s[i] < min)
            min = s[i];
    }
    return min;
```

	1	2	3	4	5
s					

Example 4

- Write an algorithm than returns the smallest value in the array s[1], s[2], …, s[n]. Use a while loop.

```
Array_min(s)
    min = s[1];
    i = 2;
    while (i ≤ n ) {
        if (s[i] < min)
            min = s[i];
        i = i + 1;
    }
    return min;
```

Example 5

- Write an algorithm that delete the ith element of an array with n elements ($1 \le i \le n$)

```
Delete(s,i)
    s[i] = s[n];
    n = n-1;
    return;
```

1	2	3	4	5	6
q	a	n	s	w	u

$i = 3$

$n = 5$

Example 6

- Write an algorithm that returns the index of the first occurrence of the value *key* in the array *s*[1], …, *s*[*n*]. If *key* is not in the array, the algorithm returns the value 0.

find_first_key{
    for *i* = 1 **to** *n*
        if (*s*[*i*] == *key*)
            return *i*
    return 0

Example 7

Algorithm Insertion_Sort (example) Insertion Sort

<u>Input</u>: $x_1, x_2, ..., x_n$

<u>Output</u>: The sorted sequence of $x_1, x_2, ..., x_n$

```
for (j = 2 to n) {      //start from the 2nd item of the unsorted list
    y = x[j]            //compare with the numbers on sorted
    i = j-1            //list from right (big) to left (small)
    while (y<x[i] and i > 0) {
        x[i+1] = x[i]
        i = i-1
    }
    x[i+1] = y
}
```
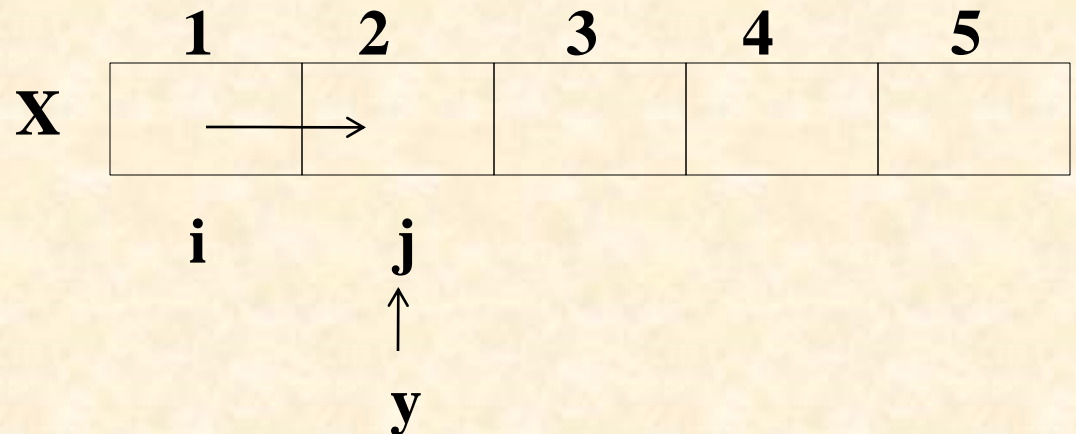
1	2	3	4	5

X

i   j

y

Example 8

- Write an algorithm to compute a^n

<div style="display: flex;">

```
exp(a,n){
    i = 1
    pow = 1
    while (i ≤ n) {
        pow = pow * a
        i = i + 1
    }
    return  pow
}
```

i	pow
1	1
2	a
3	a^2
4	a^3
n	a^{n-1}
n+1	a^n

</div>

Example 9

- Write an algorithm the returns the index of the last occurrence of the largest element in the array s[1],…s[n].

```
Find_last_largest(s)
    index = 1;
    for (i = 2 to n ) {
        if (s[i] ≥ s[index])
            index = i;
    }
    return index;
```

index	s[index]
3 1	11 11

	1	2	3	4
s	11	5	11	9

Example 10

- Write an algorithm that returns the index of the first item that is less than its predecessor in the array s[1],…,s[n]

```
Find_out_of_order(s)
    for (i = 2 to n ) {
        if (s[i] < s[i-1])
            return i;
    }
    return 0;
```
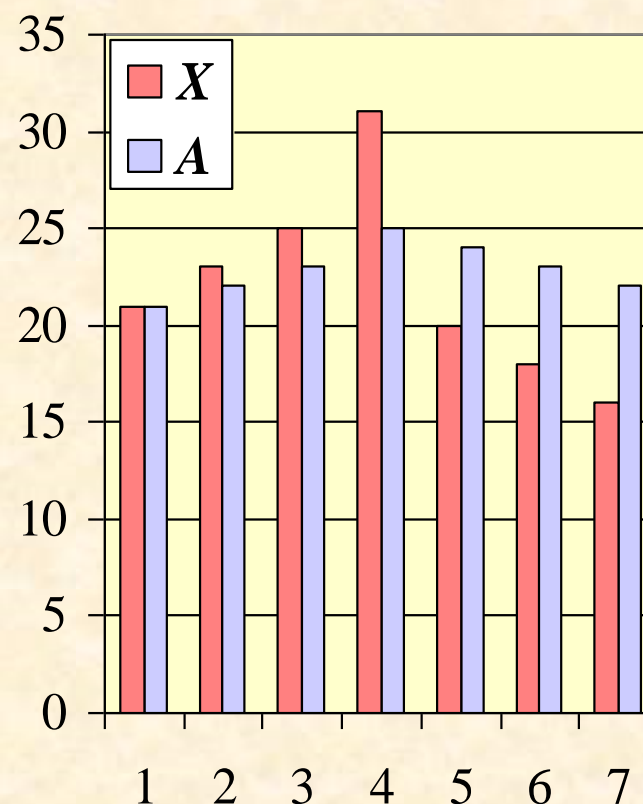
	1	2	3	4
s	11	5	11	9

	1	2	3	4
s	11	12	15	19

Example 11 :Computing Prefix Averages

- We further illustrate asymptotic analysis with two algorithms for prefix averages

- The i-th prefix average of an array X is average of the first $(i + 1)$ elements of X:

$$A[i] = (X[0] + X[1] + \ldots + X[i])/(i+1)$$

Prefix Averages (Quadratic)

♦ **The following algorithm computes prefix averages in quadratic time by applying the definition**

Algorithm *prefixAverages1*(X, n)

  Input array X **of** n **integers**

  Output array A **of prefix averages of** X

  $A \leftarrow$ **new array of** n **integers**

  for $i = 0$ **to** $n - 1$ **do**

    $s = X[0]$

    for $j = 1$ **to** i **do**

      $s = s + X[j]$

    $A[i] = s / (i + 1)$

  return A

Prefix Averages (Linear)

◆ **The following algorithm computes prefix averages in linear time by keeping a running sum**

	0	1	2
s			

Algorithm *prefixAverages2(X, n)*
  Input array *X* **of** *n* **integers**
  Output array *A* **of prefix averages of** *X*
  A ← **new array of** *n* **integers**
  s = 0
  for *i* = 0 **to** *n* − 1 **do**
    s = *s* + *X[i]*
    A[i] = *s* / (*i* + 1)
  return *A*

i	s	A[i]
0	*X[0]*	*S/1*
1	*X[0] + X[1]*	*S/2*
2	*X[0] + X[1] + X[2]*	*S/3*

Analysis of Algorithms

Input   **Algorithm**   **Output**
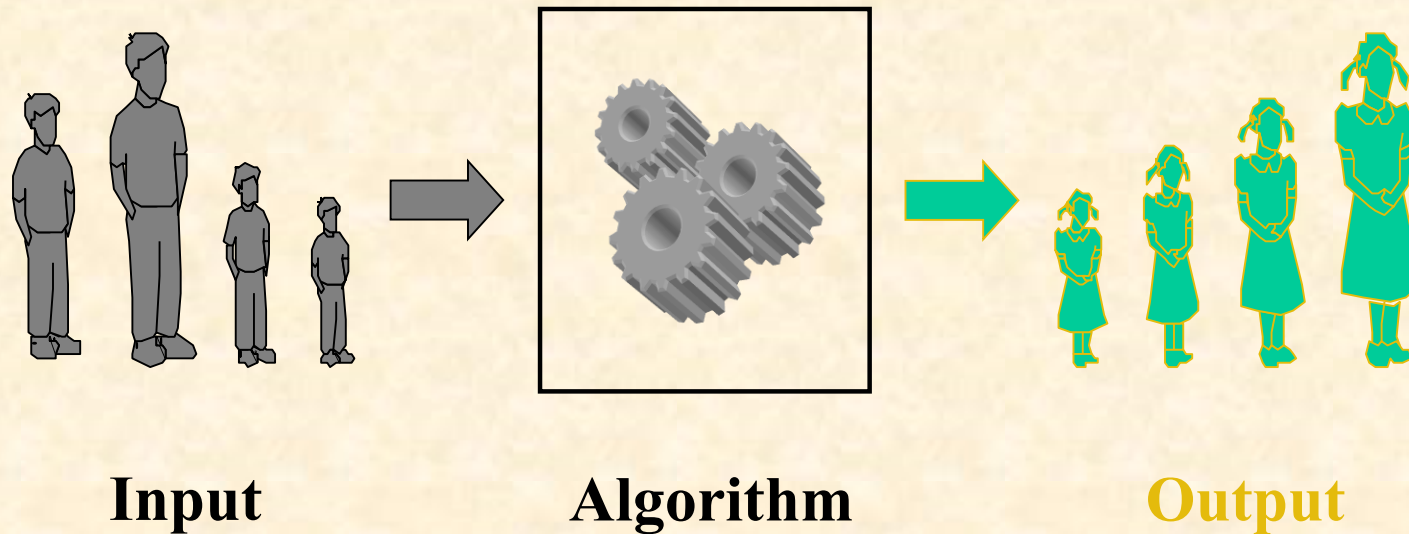
Foundations of Algorithm Analysis and Data Structures

- Algorithm Analysis:
  - How to predict an algorithm's performance
  - How well an algorithm scales up
  - How to compare different algorithms for a problem

- Data Structures
  - How to efficiently store, access, manage data
  - Data structures effect algorithm's performance

How to Measure Algorithm Performance?

- How to Measure Algorithm Performance?

  – Time efficiency: How long does it take to execute the algorithm or how many instructions does the algorithm execute?

  – Space efficiency: How much memory does the algorithm need during execution?

- **Time Efficiency is the dominant standard**.

  – Quantifiable and easy to compare
  – Often the critical bottleneck

Time Efficiency

- Measures
  - how long does it take to execute the algorithm ?
  - how many instructions does the algorithm execute?

- Approaches
  - Empirical Studies
    - write programs to implement algorithms
    - measure the running time
  - Theoretical Analysis
   - count how many instructions are executed

Empirical Studies: 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
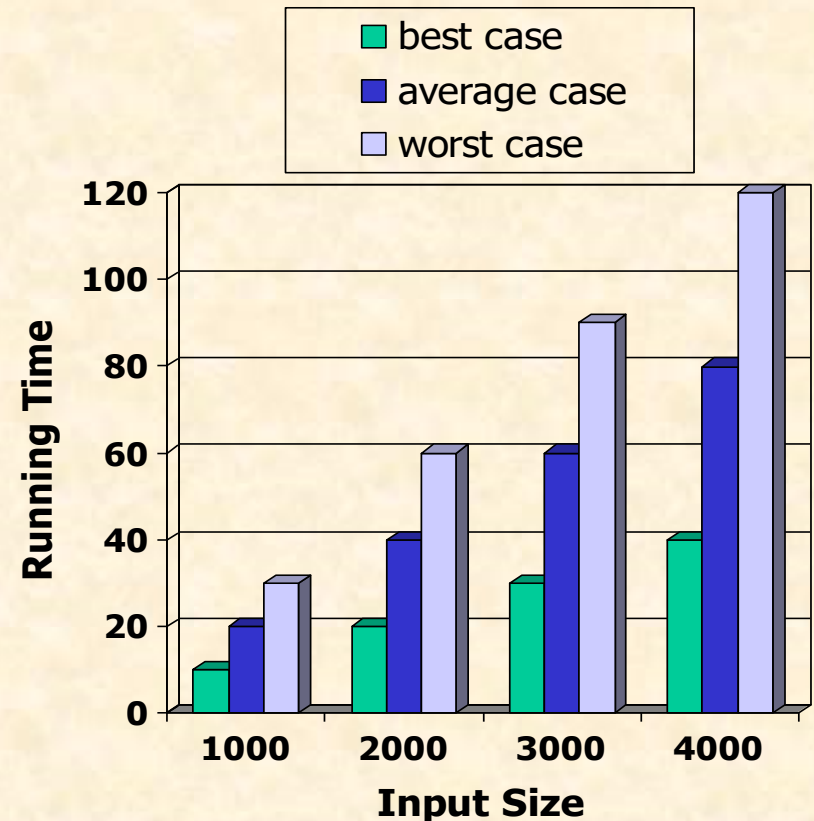
Empirical Studies: Implementation

- 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
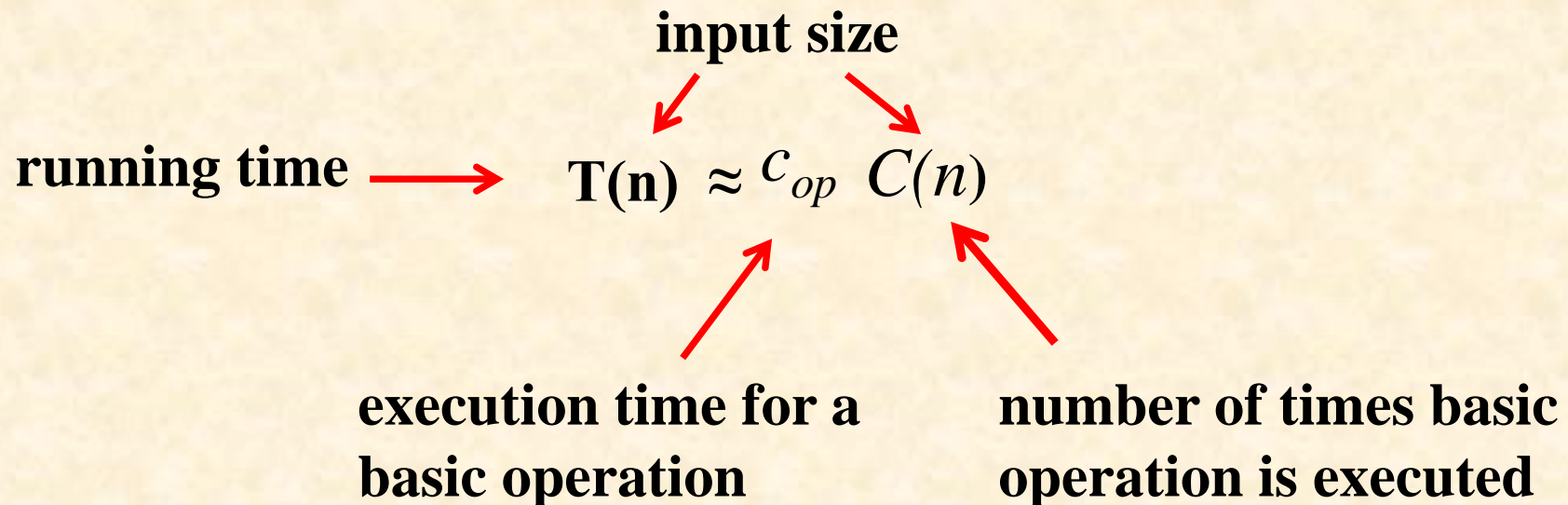- Plot the results
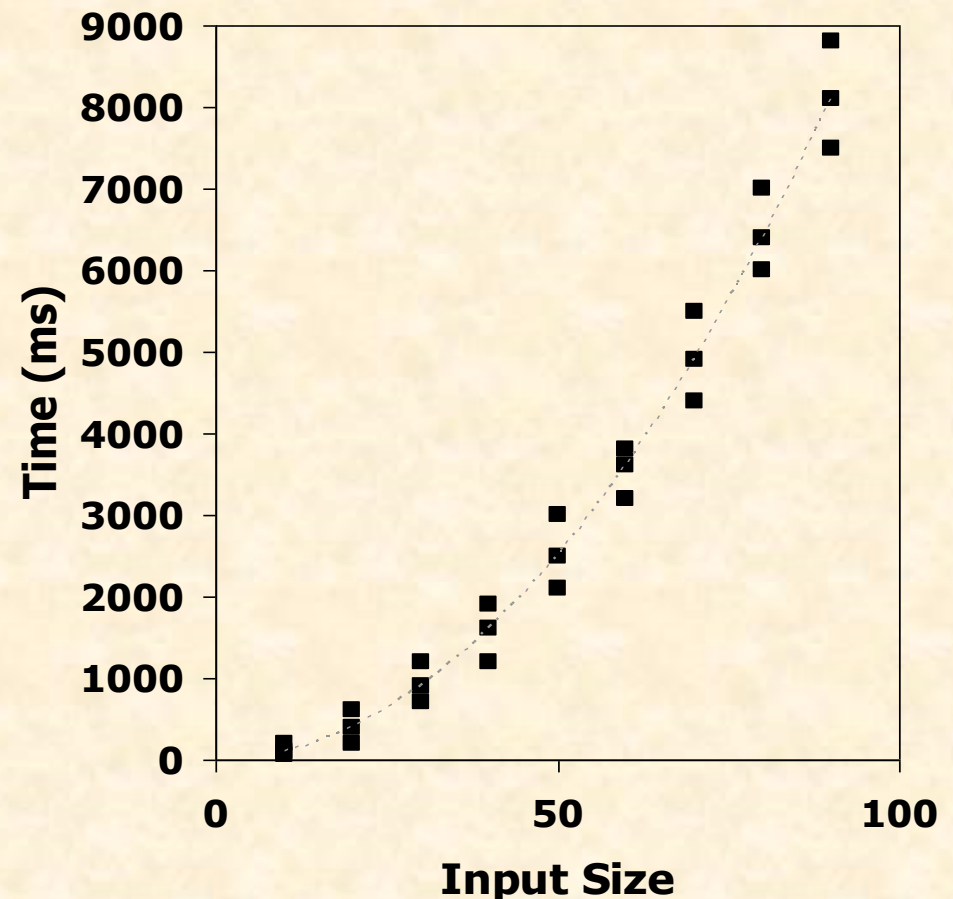
Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult and time consuming

- 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

➢ One of the main reasons for studying algorithm analysis

Theoretical analysis of time efficiency

Time efficiency is analyzed by determining the number of repetitions of the *basic operation* as a function of *input size*

- *Basic operation:* the operation that contributes most towards the running time of the algorithm.

input size

running time $\longrightarrow$ $\mathbf{T(n)} \approx c_{op}\ C(n)$

execution time for a basic operation

number of times basic operation is executed

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

---

Time Efficiency

Time efficiency can be evaluated based on three scenarios:

- Worst case: $W(n)$ – maximum number of basic operations over inputs of size n

- Best case: $B(n)$ – minimum number of basic operations over inputs of size n

- Average case: $A(n)$ – "average" over inputs of size n

Basic Operations

- *Basic operation:* the operation that contributes most towards the running time of the algorithm

- Identifiable in pseudocode

- Largely independent from the programming language

Basic operation examples

Problem	Basic Operation
Searching for key in a list of n items	Key comparison
Multiplying two matrices	Multiplication of two numbers
Sorting a set of numbers	Key comparison
Check if a given integer is a prime number	Division

	0	1	2	3	4
S	2	3	2	23	

Prime Numbers

A **prime number** (or a **prime**) is a natural number greater than 1 that has no positive divisors other than 1 and itself.

Examples: 2, 5, 7, 11, 13 …

Testing for Prime Numbers : **Trial Division**

Given an input number n, check whether any integer m from 2 to $n - 1$ evenly divides n (the division leaves no remainder). If n is not divisible by any m then it is a prime number)

For example, to test whether 5 is prime, test whether 5 is divisible by 2, or 3, or 4. Since a prime is only divisible by 1 and itself, if we reach 4 without finding a divisor, then we have proven that 5 is prime.

Counting Basic Operations

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

Algorithm *arrayMax*(**A**, **n**)	# operations
Max = *A*[0]	1
for *i* = 1 **to** *n* − 1 **do**	*n* − 1
if *A*[*i*] > *Max* **then**	*n* − 1
Max = *A*[*i*]	*n* − 1
{ increment counter *i* }	*n* − 1
return *Max*	1

Estimating Running Time

- Algorithm *arrayMax* executes n-1 basic 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
$$a\,(n-1) \leq T(n) \leq b(n-1)$$

- 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 intrinsic property of algorithm *arrayMax*

- Algorithms can be compared based on the running times, $T(n)$

Independent of hardware/software environment

Caters for all possible inputs

Principles of Algorithm Analysis

- Two algorithms, *A* and *B*, for solving a given problem.
  - Let the running times of the algorithms to be $T_a(n)$ and $T_b(n)$
  - Suppose the problem size is n_0 and $T_a(n_0) < T_b(n_0)$.
    - Then algorithm *A* is better than algorithm *B* for **problem size n_0**. (not good enough)

❖ If $T_a(n) < T_b(n)$ for all n, $n \geq n_0$, then algorithm *A* is better than algorithm *B* regardless of the problem size.

- Note: Our primarily concern is to estimate the time of an algorithm instead of computing its exact time
  - a useful measurement of time can be obtained by counting the ***fundamental, dominating steps*** of the algorithm Slide 20
  - E.g. counting the number of comparisons in a sorting algo.

Ver 1.0   EE2008 Data Structure and Algorithms   *Intro/Page 98*

- Let
  - c_{op} = execution time of an algorithm's basic operation on a computer
  - $C(n)$ = number of times the basic operation is executed for this algorithm **Operation Count**
  - $T(n)$ = running time of this algo. on the computer

    $$T(n) \approx c_{op} \, C(n)$$

- Assume that for an algorithm, the operation count $C(n) = \frac{1}{2} n(n-1)$. How much longer will the algo. run if its input size is doubled?

  $$C(n) = \tfrac{1}{2} \, n(n-1) \approx \tfrac{1}{2} \, n^2$$

  $$\frac{T(2n)}{T(n)} = \frac{c_{op} C(2n)}{c_{op} C(n)} \approx \frac{\dfrac{1}{2}(2n)^2}{\dfrac{1}{2}(n)^2} = 4$$

  <div style="border:2px solid blue">

  Order of Growth:

  How fast does the operation count increase as the input size increases?

  </div>

  When the input size is doubled, the run time is 4 times longer.

- For algorithm analysis, emphasize on the operation count's **order of growth** for large input sizes (*asymptotic behaviour*)
  - Note: the difference in running times on small inputs cannot really distinguish efficient algorithms from inefficient ones
    - Interested in large values of input, n
- To compare and rank the order of growth (for comparing the efficiency of different algorithms), three notations have been defined:
  - O (big oh), Ω (big omega) and Θ (big theta)

**y=g(n) (algo B)**

y=f(n) (algo A)
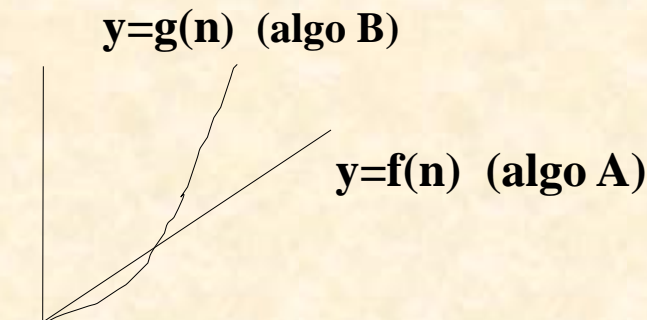
Worst-case analysis

Compare order growth of operations count under worst-case scenario

Asymptotic Analysis

Asymptotic Notation

- **In comparing algorithms, in general, we consider the** *asymptotic* **behavior of the two algorithms for large problem sizes, under worst-case.**

- *Big-Oh* notation: a notation used for characterizing the asymptotic behavior of functions.
  - gives an asymptotic upper bound for a function to within a constant factor Big-Oh Asymptotic Notation   Big-Oh   Asymptotic Notation

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

  - The growth rate of $f(n)$ is less than or equal to the growth rate of $g(n)$

  - $g(n)$ is an upper bound on $f(n)$
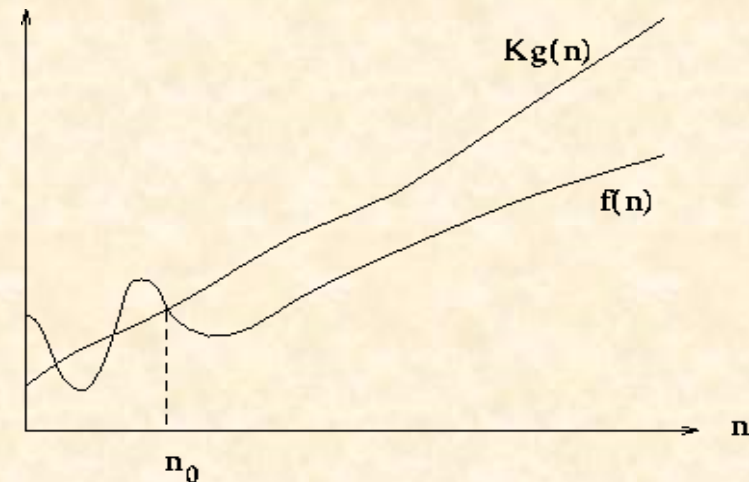
Big-Oh Notation

- ***Big-Oh*** notation
  - gives an asymptotic upper bound for a function to within a constant factor

Asymptotic Notation   Big-Oh   Asymptotic Notation   Big-Oh
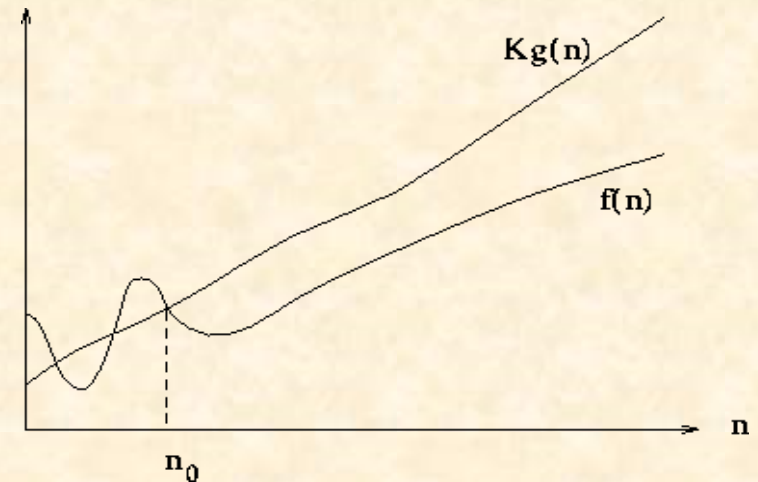
Compare the order of growth of operation count for large inputs

Definition: Given non-negative functions $f(n)$ and $g(n)$, we say that

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

if there exists an integer n_0 and a constant $k>0$ such that
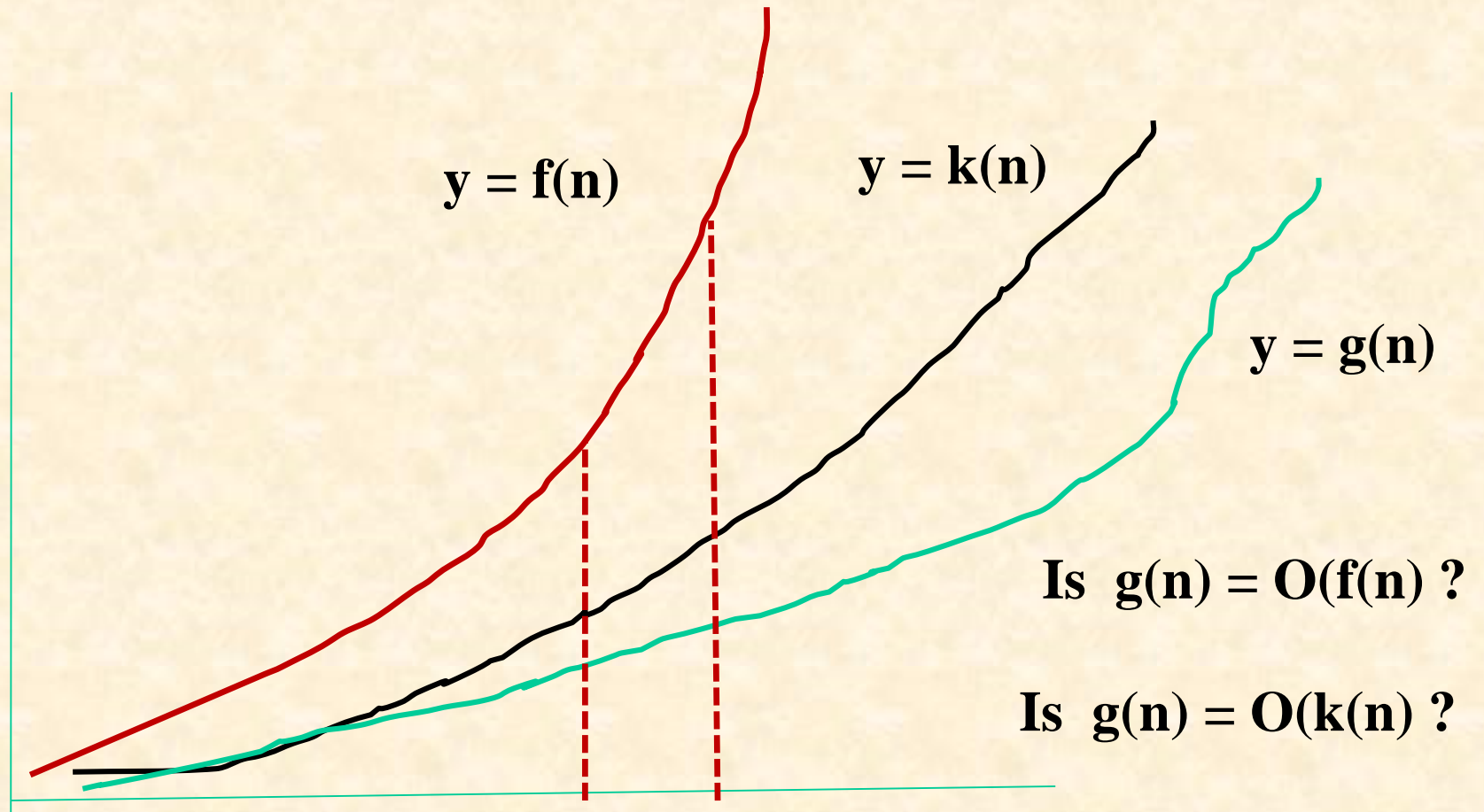
$$f(n) \leq kg(n) \text{ for all integers } n \geq n_0$$

- $f(n)=O(g(n))$: $f(n)$ **is of order at most $g(n)$ or $f(n)$ is big oh of $g(n)$**

$f(n)$ grows no faster than $g(n)$ for large n

Big-Oh

y = f(n)

y = k(n)

y = g(n)

Is g(n) = O(f(n)) ?

Is g(n) = O(k(n)) ?

EE2008 Data Structure and Algorithms

Big-Oh

- Example

$$\frac{T(2n)}{T(n)} = \frac{c(2n)^2}{c(n)^2} = 4$$

  - $f(n) = O(n^2)$

    – Growth rate of f(n) is less than or equal to n^2

    – Time or number of operations does not exceed $c.n^2$ on any input of size n (n suitably large).

    – So, the time or number of operations is expected to quadruple each time n is doubled

    – $O(n^2)$: reads "order n-squared" or "Big-oh n-squared"

Example

- Example: $2n + 10$ is
  $O(n)$ Big-Oh: Example 1 Example

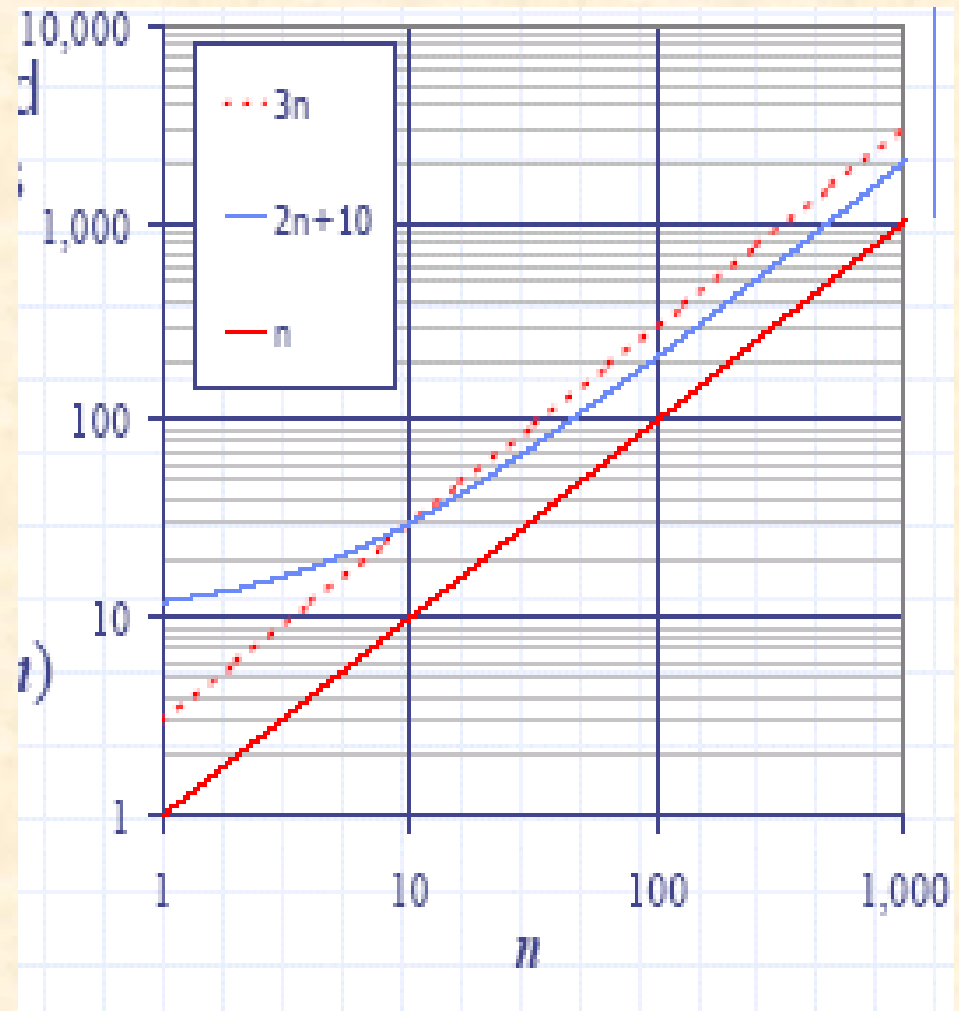$$2n + 10 \leq kn \quad \text{for } n \geq n_0$$

$$10 \leq (k - 2)\, n$$

$$n \geq 10/(k - 2)$$

Pick $k = 3$ and $n_0 = 10$, so
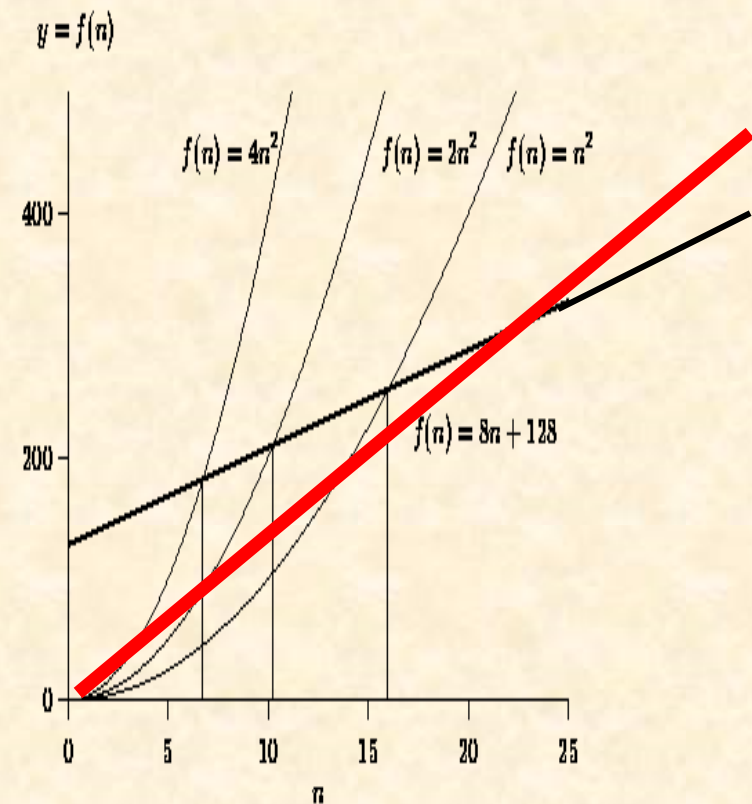
$$2n + 10 \leq 3n \quad \text{for } n \geq 10$$

- Example: Given $f(n)=8n+128$ and $g(n) = n^2$. Is $f(n) = O(g(n))$?

$$\mathbf{f(n) = O(n^2)}$$

$$\mathbf{f(n) = O(n)} \longleftarrow \textbf{Best answer,}$$
$$\textbf{asymptotically tight}$$

- We need to find an integer n_0 and a constant $k>0$ such that for all integers $n \geq n_0$, $f(n) \leq kn^2$.

- We must get $kn^2 - 8n - 128 \geq 0$
  - If we set $k=1$, we have
    $(n-16)(n+8) \geq 0$;
    hence, we need $(n-16) \geq 0$,
    i.e. $n_0 = 16$, k=1.

- Of course, there are many other values of k and n_0 that will do. For example, $k=2$ and $n_0 = 11$.

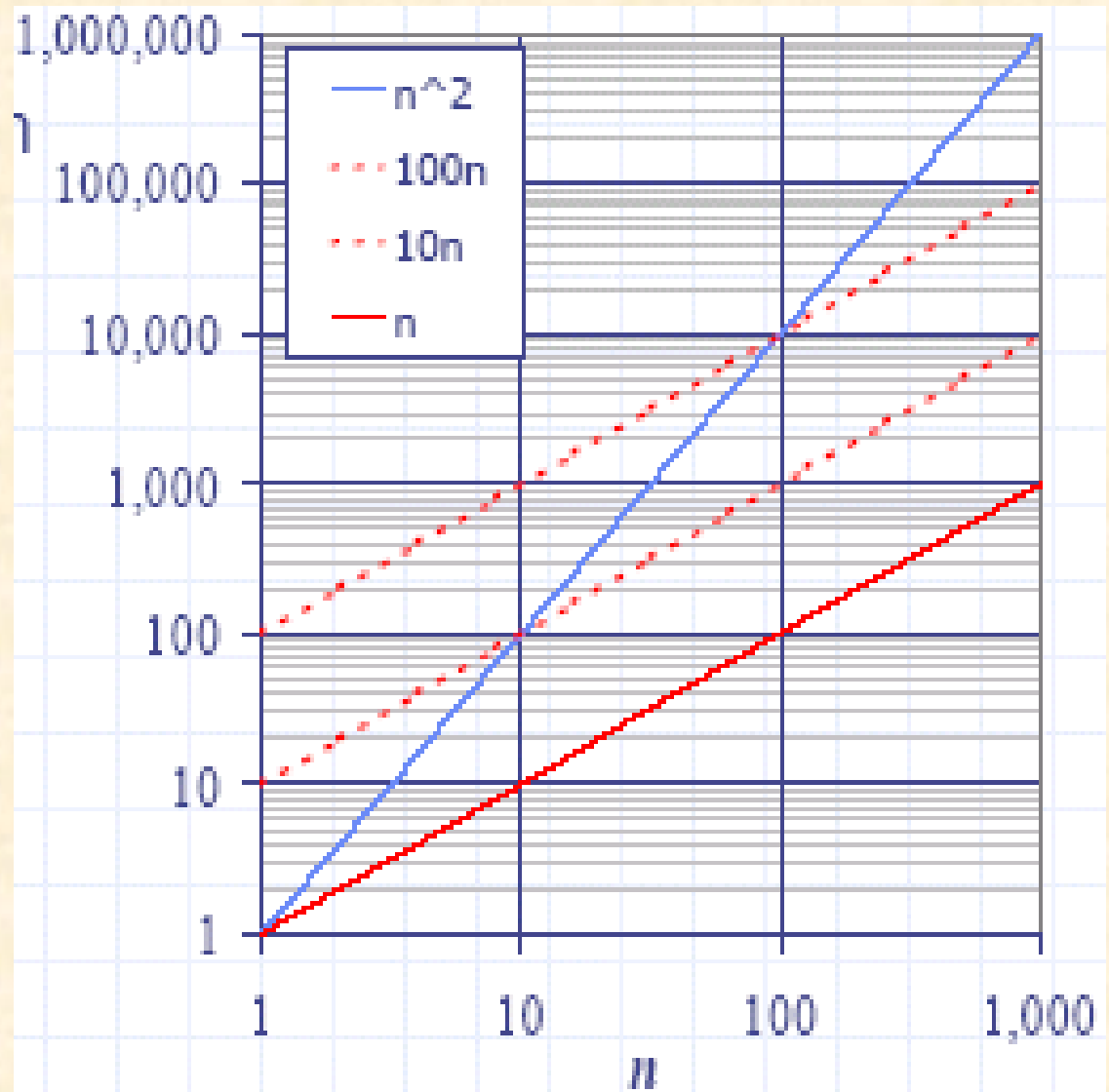- If $g(n) = n$, prove that $f(n) = O(g(n))$.

Example

- Example: the function n^2 is not $O(n)$

    $$n^2 \leq kn$$

    $$n \leq k$$

    – The above inequality cannot be satisfied since k must be a constant

More Big-Oh Examples

- $7n$-2
  - $7n$-2 is $O(n)$
  - need $k > 0$ and $n_0 \geq 1$ such that $7n$-$2 \leq kn$ for n $\geq n_0$
  - this is true for $k = 7$ and $n_0 = 1$
- $3n^3 + 20n^2 + 5$ Big Oh : Example 5
  - $3n^3 + 20n^2 + 5$ is $O(n^3)$
  - need $k > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq kn^3$ for $n \geq n_0$
  - this is true for $k = 4$ and $n_0 = 25$
- $3\ lg\ n + 5$ Big Oh : Example 6   More Big-Oh Examples
  - $3\ lg\ n + 5$ is $O(lg\ n)$
  - need $k > 0$ and $n_0 \geq 1$ such that $3\ lg\ n + 5 \leq k\ lg\ n$ for n $\geq n_0$
  - this is true for $k = 8$ and $n_0 = 2$

Big Oh : Example

- Given f(n) = $3n^3 + 20n^2 + 5$ show that f(n) = $O(n^3)$
- Need to find constants k and n_0 such that

$$3n^3 + 20n^2 + 5 \leq kn^3 \text{ for } n \geq n_0$$
$$(k-3)n^3 - 20n^2 - 5 \geq 0 \text{ for } n \geq n_0$$

Let k = 4:

$$n^3 - 20n^2 - 5$$
$$\geq \quad n^3 - 20n^2 - 5n^2$$
$$= n^2(n-25)$$
$$\geq 0 \qquad \text{for } n \geq 25$$

Hence by choosing k=4 and $n_0 = 25$, we have

$$3n^3 + 20n^2 + 5 \leq kn^3 \text{ for } n \geq n_0$$

Another Proof

- $3n^3 + 20n^2 + 5$

  $\leq 3n^3 + 20n^3 + 5n^3 \qquad$ for $n \geq 1$

  $= 28n^3$

Hence by choosing k=28 and $n_0 = 1$, we have

$$3n^3 + 20n^2 + 5 \leq kn^3 \quad \text{for all } n \geq n_0$$

So, $3n^3 + 20n^2 + 5 = O(n^3)$

Big Oh : Example

- Given $f(n) = 3 \lg n + 5$ show that $f(n) = O(\lg n)$
- Need to find constants k and n_0 such that

$$3 \lg n + 5 \le k \lg n \quad \text{for } n \ge n_0$$

$$(k\text{-}3) \lg n - 5 \ge 0 \text{ for } n \ge n_0$$

Let $k = 8$:

$$5 \lg n - 5$$

$$\ge \quad 0 \quad \textit{for all } n \ge 2$$

$$\boxed{\textcolor{red}{\lg n \ge 1 \quad \text{for } n \ge 2}}$$

Hence by choosing $k=8$ and $n_0 = 2$, we have

$$3 \lg n + 5 \le k \lg n \text{ for } n \ge n_0$$

Big-Oh Rules

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
$$f(n) = c_d n^d + c_{d-1} n^{d-1} + \ldots + c_1 d + c_0$$

  1. Drop lower-order terms

  2. Drop constant factors Polynomial.pptx

- Use the smallest possible class of functions

  – Say "100n is $O(n)$" instead of "100n is $O(n^2)$"

- Use the simplest expression of the class

  – Say "3n + 5 is $O(n)$" instead of "3n + 5 is $O(3n)$"

Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function

- The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$

- We can use the big-Oh notation to rank functions according to their growth rate

Seven Important Functions

- The amount of time required to execute an algorithm usually depends on the input size, n

- Seven functions that often appear in algorithm analysis:
  - Constant ≈ 1 *constant.pptx*
  - Logarithmic $\approx \log n$
  - Linear $\approx n$
  - N-Log-N $\approx n \log n$
  - Quadratic $\approx n^2$
  - Cubic $\approx n^3$
 - Exponential $\approx 2^n$
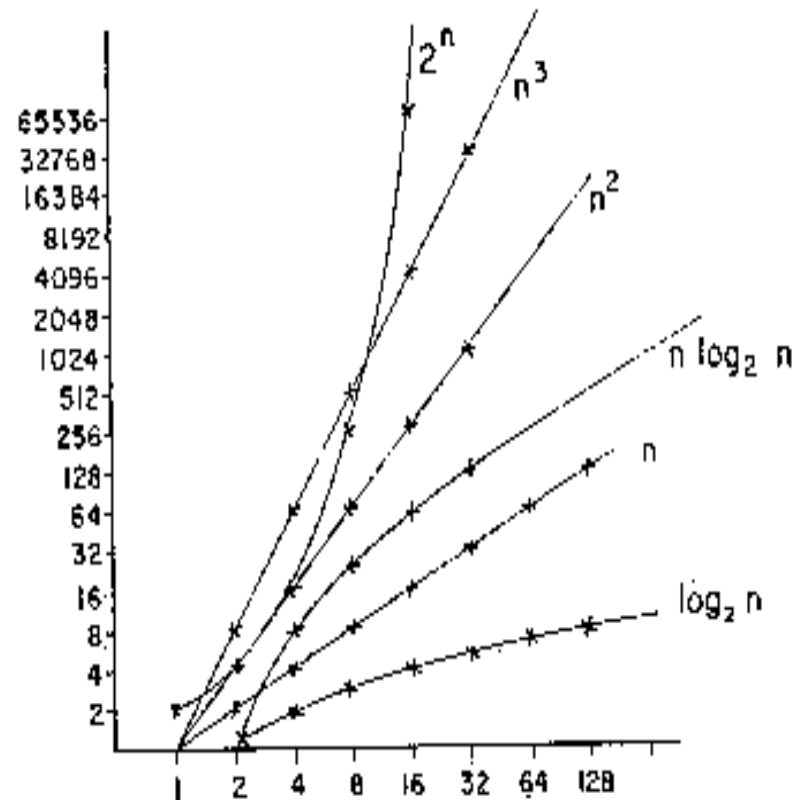
- The following table shows the running times computed for a very conservative scenario.
  - We assume that the constant, c, is one cycle of a 100 MHz clock. This table shows the running times we can expect even if only one instruction is done for each element of the input

	$n=1$	$n=8$	$n= 1k$	$n= 1024k$
$O(1)$	10ns	10ns	10ns	10ns
$O(\lg n)$	10ns	30ns	100ns	200ns
$O(n)$	10ns	80ns	1.02 μs	10.05ms
$O(n\lg n)$	10ns	240ns	10.2 μs	210ms
$O(n^2)$	10ns	640ns	102 μs	3.05hours
$O(n^3)$	10ns	5.12μs	10.7s	365years
$O(2^n)$	10ns	2.56μs	10^{293}years	$10^{10^{15}}$years

Constant time C ==> $C \leq C(1)$ for $n \geq 1$ ==> $C = O(1)$

H

Asymptotic notation
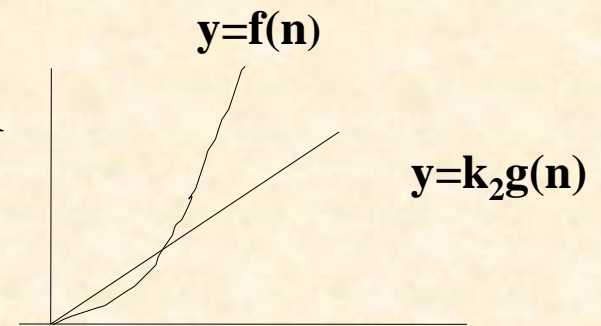
- $f(n) = \Omega(g(n))$
    - $f(n)$ is of order **at least** $g(n)$ or $f(n)$ is omega of $g(n)$ if there exist constants k_2 and n_2 such that $f(n) \geq k_2 g(n)$ for all $n \geq n_2$

- $f(n) = \Theta(g(n))$
    - $f(n)$ is of order $g(n)$ or $f(n)$ is theta of $g(n)$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

- Hence,
    - if $f(n) = O(g(n))$, g is an asymptotic upper bound for f.
    - if $f(n) = \Omega(g(n))$, g is an asymptotic lower bound for f.
    - if $f(n) = \Theta(g(n))$, g is an asymptotic tight bound for f.

Exercise: Show that for $f(n) = 8n + 128$, $g(n) = n$, $f(n) = \Omega(g(n))$.   Big-Omega
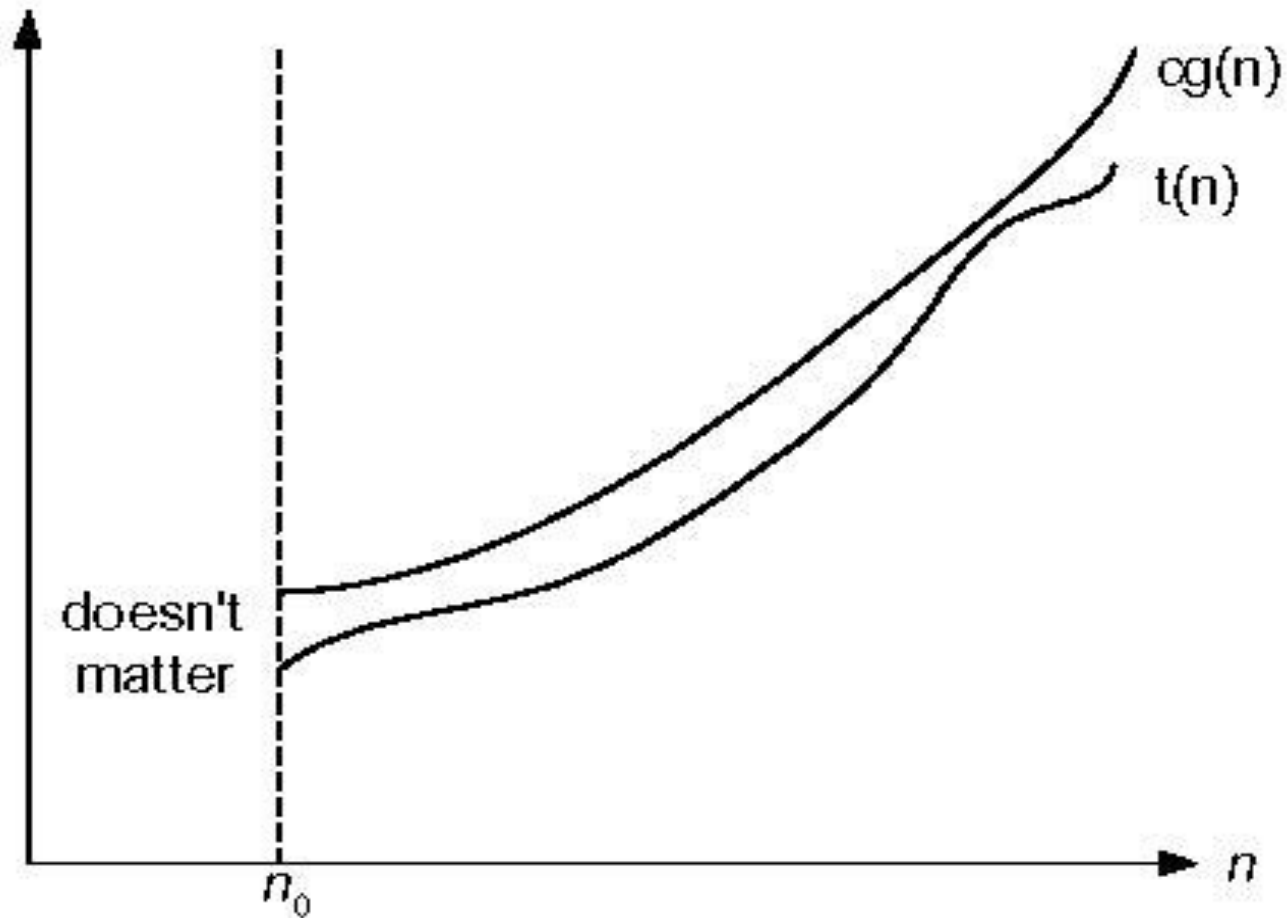
Asymptotic notation

Big-oh

Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$
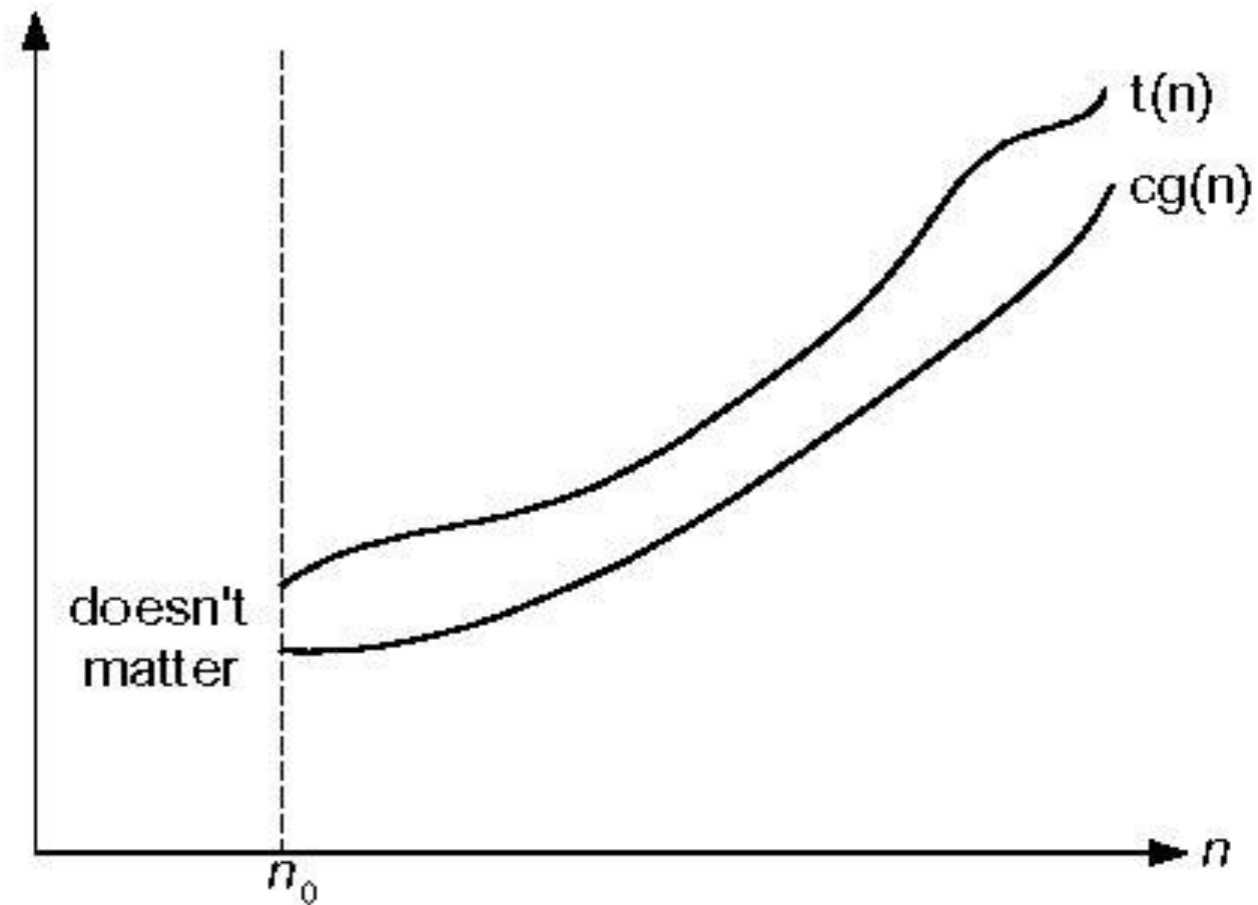
Big-omega

Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$
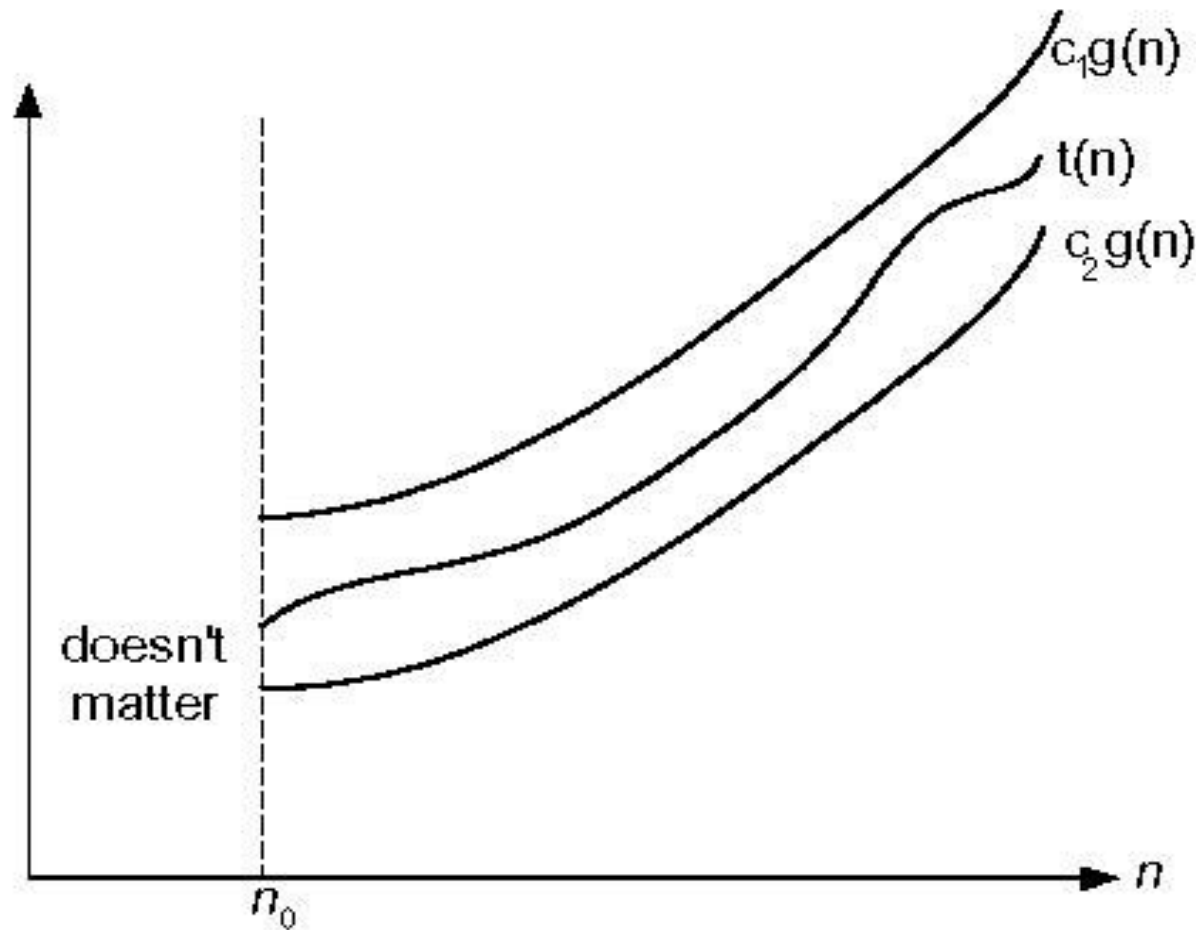
Big-theta

Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Example

Given $f(n)=60n^2+5n+1$, $g(n)=n^2$, prove that $60n^2+5n+1 = \Theta(n^2)$

Proof:

$\quad 60n^2+5n+1 \leq 60n^2+5n^2+n^2 = 66n^2 \qquad$ for all $n \geq 1$

Hence,

$\quad$ we can take $k_1 = 66$, and $n_1=1$, and conclude that $f(n) = O(n^2)$

Since $60n^2+5n+1 \geq 60n^2$, for all $n \geq 1$,

$\quad$ we can take $k_2 = 60$, and $n_2=1$, and conclude that $f(n) = \Omega(n^2)$

Based on the above, we have $60n^2+5n+1 = \Theta(n^2)$

Example

Given $f(n)=2n + 3 \lg n$, $g(n)=n$, prove that $2n + 3 \lg n = \Theta(n)$

Proof:

$2n + 3 \lg n \le 2n+3n = 5n$ for all $n \ge 1$

Hence,

we can take $k_1 =5$, and $n_1=1$, and conclude that $f(n) = O(n)$

Since $2n + 3 \lg n \ge 2n$, for all $n \ge 1$,

we can take $k_2 = 2$, and $n_2=1$, and conclude that $f(n) = \Omega(n)$

Therefore, we have $2n + 3 \lg n = \Theta(n)$

Example

Given $f(n)=1+2+\ldots+n$, show that $f(n) = \Theta(n^2)$

Proof:

$\quad 1+2+\ldots+n \leq n+n+\ldots+n = n \cdot n = n^2$ $\qquad$ for all $n \geq 1$

Hence,

$\quad$ we can take $k_1 =1$, and $n_1=1$, and conclude that $f(n) = O(n^2)$

Since $1+2+\ldots+n = n(n+1)/2$, for all $n \geq 1$,

$\quad n(n+1)/2 = (n^2+n)/2 \geq n^2/2$

$\quad$ we can take $k_2 = \frac{1}{2}$, and $n_2=1$, and conclude that $f(n) = \Omega(n^2)$

Therefore, we have $f(n) = \Theta(n^2)$

Properties of Asymptotic

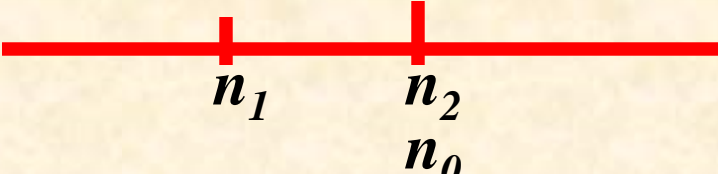Suppose we know that $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$

- What can we say about the asymptotic behavior of the *sum* and the product of $f_1(n)$ and $f_2(n)$?

- Theorem 3.1: $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$ Slide 145
- Theorem 3.2: $f_1(n) \times f_2(n) = O(g_1(n) \times g_2(n))$ Thm3_2.pptx

- Consider the functions $f_1(n) = n^3 + n^2 + n + 1 = O(n^3)$ and
$f_2(n) = n^2 + n + 1 = O(n^2)$

  - By Theorem 3.2 , the asymptotic behavior of the product $f_1(n) \times f_2(n)$ is $O(n^3 \times n^2) = O(n^5)$.

Ver 1.0 EE2008 Data Structure and Algorithms *Intro/Page 124*

Proof of theorem 3.1: Properties of Asymptotic

- Given that $f_1(n) = O(g_1(n))$ & $f_2(n) = O(g_2(n))$
  - So by definition, there exist two integers, n_1 and n_2, and two constants k_1 and k_2 such that

  $f_1(n) \leq k_1\, g_1(n)$ for $n \geq n_1$ and $f_2(n) \leq k_2\, g_2(n)$ for $n \geq n_2$

- Let $n_0 = \max(n_1, n_2)$

  $n \geq n_0 \Rightarrow n \geq n_1$

  $n \geq n_0 \Rightarrow n \geq n_2$

- $k_0 = 2\max(k_1, k_2)$ ⟵ $k_1 \leq 1/2k_0$ $k_2 \leq 1/2k_0$

  $k_0 \geq 2k_1$?

- Note that

  $k_0 \geq 2k_2$?

  $f_1(n) \leq k_1\, g_1(n)$ for $n \geq n_0$

  and $f_2(n) \leq k_2\, g_2(n)$ for $n \geq n_0$

So, $f_1(n) + f_2(n)$ $\leq k_1\, g_1(n) + k_2\, g_2(n)$ for $n \geq n_0$

$\leq k_0\, (g_1(n) + g_2(n))/2$ $g_1(n) \leq \max(g_1(n), g_2(n))$

$\leq k_0\, \max(g_1(n), g_2(n))$ $g_2(n) \leq \max(g_1(n), g_2(n))$

- Thus $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$ $g_1(n) + g_2(n) \leq 2\max(g_1(n), g_2(n))$

Proof of theorem 3.1:

- By definition, there exist two integers, n_1 and n_2, and two constants k_1 and k_2 such that

$$f_1(n) \le k_1 \, g_1(n) \text{ for } n \ge n_1 \quad \text{and} \quad f_2(n) \le k_2 \, g_2(n) \text{ for } n \ge n_2$$

- Let $n_0 = \max(n_1, n_2)$, $k_0 = 2 \max(k_1, k_2)$, consider the sum for $f_1(n) + f_2(n)$ for $n \ge n_0$,

$$f_1(n) + f_2(n) \le k_1 \, g_1(n) + k_2 \, g_2(n) \quad \text{for } n \ge n_0$$

$$\le k_0 \, (g_1(n) + g_2(n))/2$$

$$\le k_0 \, \max(g_1(n), g_2(n))$$

- Thus $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$ *Properties of Asymptotic*

Proof of theorem 3.2:

- By definition, there exist two integers, n_1 and n_2, and two constants k_1 and k_2 such that

$$f_1(n) \leq k_1\, g_1(n) \text{ for } n \geq n_1 \quad \text{and} \quad f_2(n) \leq k_2\, g_2(n) \text{ for } n \geq n_2$$

- Let $n_0 = \max(n_1, n_2), \quad k_0 = k_{1*}\, k_2,$
- Note that

$$f_1(n) \leq k_1\, g_1(n) \qquad \text{for } n \geq n_0$$
$$\text{and} \quad f_2(n) \leq k_2\, g_2(n) \qquad \text{for } n \geq n_0$$

$$
\begin{aligned}
f_1(n) * f_2(n) \quad &\leq k_1\, g_1(n) * k_2\, g_2(n) \qquad \text{for } n \geq n_0 \\
&= k_1\, k_2\, g_1(n) g_2(n) \\
&= k_0\, g_1(n) g_2(n)
\end{aligned}
$$

- Thus $f_1(n) * f_2(n) = O(g_1(n) * g_2(n))$

Algorithm analysis

- Concern primarily with the running time and memory space needed to execute a program
- Factors that affect the running time of a program.
  - The algorithm used, the input data, and the computer system
  - The performance of a computer is determined by
    - processor used (type and speed),
    - memory available (cache and RAM) and disk access speed,
    - the programming language used, and
    - the computer operating system software.
- A detailed analysis which takes all of these factors into account is a very difficult and time-consuming
  - only consider a simple model of algorithm independent of the others since we are concerned with the estimating the time of an algorithm

Algorithm analysis - example

- A simple algorithm:

  for $i = 1$ to n

     for $j = 1$ to n

        $x = x+1$

i	j	No of time "x=x+1" executed
1	1	1
	2	1
	3	1
	n	1

- What is the number of times the statement $x = x+1$ being executed?
  - The running time of the inner loop $j = 1$ to n is $O(n)$
  - The running time of the outer loop $i = 1$ to n is $O(n)$
  - As they are nested loops, the total time is $O(n^2)$

- Example: $S_j = \sum_{i=0}^{j} a_i$ $S_0 = \sum_{i=0}^{0} a_i = a_0$ $S_1 = \sum_{i=0}^{1} a_i = a_0 + a_1$

- An algorithm to compute the series of summations is given in following:

		j	number of executions of inner loop
1	for $j = 0$ to n-1		
2	$\quad$ sum = 0	0	1
3	$\quad$ for $i = 0$ to j	1	2
4	$\qquad$ sum = sum + a[i]	2	3
5	$\quad$ s[j] = sum		
		n-1	n

- The inner-most loop (steps 3 and 4) is executed for 1 to n times, i.e. $1 + 2 + \ldots + n = n(n+1)/2$ times   **n(n+1)/2 = 1/2[n² + n]**

- Therefore, the total running time of the program is $O(n^2)$.

General Plan for Algo. Analysis

1. Decide on parameter n indicating *input size*

2. Identify algorithm's *basic operation*

3. Determine *worst* case for input of size n

   – May also need to determine the *average* and *best* cases

4. Set up a sum expressing the number of times the algo's basic operation is executed

5. Simplify the sum using standard formulas and rules to determine big-Oh of the algo's running time

General Rules to determine Running Time

1. For loops

    – The running time of the statements inside the loop times the number of iterations

2. Nested loops

    – The running time of the statements inside the nested loop times the product of the sizes of all the loops

    – An example which is of $O(n^2)$

    for i = 1 to n

        for j = 1 to n

            k = k +1

---

3. Consecutive statements
   - Simply add the running time
   - Example: the fragment has $O(n)$ work followed by $O(n^2)$, which has running time being $O(n^2)$

     ```
     for i = 1 to n
       a[i] = 0           ←——— O(n)
     for i = 1 to n
       for j = 1 to n     ←——— O(n²)
         a[i] = a[j]+i+j
     ```

     $$k_1\, n + k_2\, n^2 = O(n^2)$$

   - Note: actually only the max. is counted

4. If (condition) S1
   else S2
   - The running time is the larger of the running times of S1 and S2

Example - Fibonacci numbers

- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

$$f_n = f_{n-1} + f_{n-2}, \ n \geq 2 \qquad f_0 = 0, \ f_1 = 1$$

- Algorithm 1: fibo_1(n)

      previous = 0
      result = 1
      for i = 2 to n
          fib = result + previous
          previous = result
          result = fib
      return (fib)

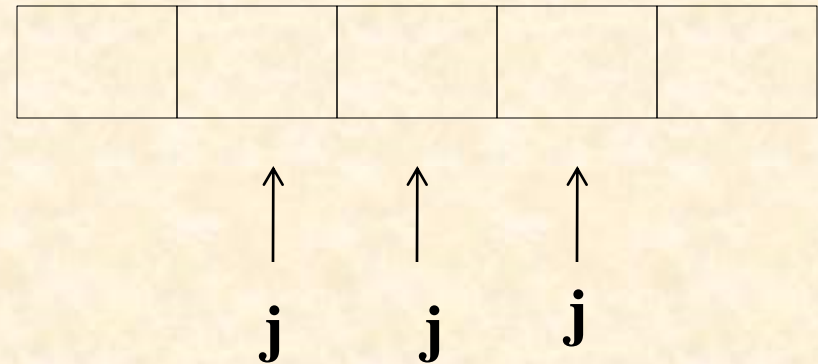| i	f_{i-2} previous	f_{i-1} result	f_i fib
2	0	1	1
3	1	1	2
4	1	2	3
5	2	3	5

- Running time: $O(n)$

Example - Insertion sort

```
for  (j = 2 to n) {          //start from the 2nd item of the unsorted list
    y = x[j]                 //compare with the numbers on sorted
    i = j-1                  //list from right (big) to left (small)
    while (y<x[i] and i > 0) {
            x[i+1] = x[i]
            i = i-1
    }
    x[i+1] = x
}
```

j   j   **j**

- while loop: executed for 1 to *n*-1 time

$$\sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} \approx \frac{n^2}{2} = O(n^2)$$

- Running time: $O(n^2)$

j	Max number of comparisons
2	1
3	2
4	3
n	n-1

Example: Given 2 *n*-by-*n* matrices, *A* and *B*, find the time efficiency of the algo. for computing their product *C=AB*.

- Note: $C[i,j] = A[i,0]B[0,j] + \ldots + A[i,k]B[k,j] + A[i,n-1]B[n-1,j]$

  for $0 \le i,j \le n-1$

ALGORITHM $MatrixMultiplication(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$
//Multiplies two *n*-by-*n* matrices by the definition-based algorithm
//Input: Two *n*-by-*n* matrices *A* and *B*
//Output: Matrix $C = AB$
for $i \leftarrow 0$ **to** $n-1$ **do**
    for $j \leftarrow 0$ *to* $n-1$ *do*
        $C[i, j] \leftarrow 0.0$
        for $k \leftarrow 0$ **to** $n-1$ **do**
            $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$
return C

k	C[i,j]
0	A[i,0]*B[0,j]
1	A[i,0]*B[0,j] + A[i,1]*B[1,j]
2	A[i,0]*B[0,j] + A[i,1]*B[1,j] + A[i,2]*B[2,j]

Matrix Multiplication

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x & p \\ y & q \\ z & r \end{bmatrix}$$

$$= \begin{bmatrix} ax+by+cz & ap+bq+cr \\ dx+ey+fz & dp+eq+fr \\ gx+hy+iz & gp+bq+ir \end{bmatrix}$$

$$\text{row}_i(A) \cdot \text{col}_j(B) = \sum_{k=1}^{n} a_{ik} b_{kj}$$

- Note that the basic operation is the multiplication at the most inner loop (with index k)
  - The number of operations to get a specific value for C[i,j] is

$$\sum_{k=0}^{n-1} 1$$

- The total number of operations is

$$\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3$$

- The running time of the algo. is $O(n^3)$

- Example: Maximum Subsequence Sum Problem
- Given n integers, $A_1, A_2, \ldots, A_n$, find the max value of $\sum_{k=i}^{j} A_k$
  - E.g. For -2, 11, -4, 13, -5, -2, the answer is 20 (A_2 to A_4)
  - Add the elements in an array by making multiple passes

```
maxSum = 0
for i = 1 to n
    for j = i to n
        thisSum = 0
        for k = i to j
            thisSum = thisSum + A[k]
        if thisSum > maxSum
            maxSum = thisSum
```

i	j	thisSum
1	1	-2
	2	-2+11
	3	-2+11+(-4)
	4	-2+11+(-4)+13
	5	-2+11+(-4)+13+(-5)
	6	-2+11+(-4)+13+(-5)+(-2)
2	2	-11
	3	-11+(-4)

- **The total number of operations is** $\sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=i}^{j}1$

Given n integers, $A_1, A_2, \ldots, A_n$, find the max value of $\sum\limits_{k=i}^{j} A_k$

E.g. -2, 11, -4, 13, -5, -2

- -2
- -2 + 11 = 9
- -2 + 11 + (-4) = 5
- -2 + 11 +(-4) + 13 = 18
- -2 + 11 + (-4) + 13 + (-5) = 13
- -2 + 11 + (-4) + 13 +(-5) + (-2) = 11
- -11
- 11 + (-4) = 7
- $\boxed{11 + (-4) + 13 = 20}$
- 11 + (-4) + 13 + (-5) = 15
- 11 + (-4) + 13 + (-5) + (-2) = 13

- -4
- (-4) + 13 = 9
- (-4) + 13 + (-5) = 4
- (-4) + 13 + (-5) + (-2) = 2
- 13
- 13 + (-5) = 8
- 13 + (-5) + (-2) = 6
- -5
- (-5) + (-2) = -7
- -2

h

$$\sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=i}^{j}1$$

$$\sum_{k=i}^{j}1 = j-i+1 \qquad \text{H}$$

$$\sum_{j=i}^{n}(j-i+1) = 1+2+...+(n-i+1) = \frac{(n-i+1)(n-i+2)}{2}$$

$$\sum_{i=1}^{n}\frac{(n-i+1)(n-i+2)}{2} = \frac{n^3+3n^2+2n}{6} \qquad \text{H}$$

- **The running time of the algo. is $O(n^3)$** Maxsubsequence_analysis.pptx

$$\sum_{i=1}^{n} \frac{(n-i+1)(n-i+2)}{2} = \sum_{i=1}^{n} \frac{n^2 - ni + 2n - ni + i^2 - 2i + n - i + 2}{2}$$

$$= \sum_{i=1}^{n} \frac{n^2 - 2ni + 3n + i^2 - 3i + 2}{2}$$

$$= \frac{n^3 - 2n\sum_{i=1}^{n} i + 3n^2 + \sum_{i=1}^{n} i^2 - 3\sum_{i=1}^{n} i + 2n}{2}$$

$$= \frac{n^3 - 2n\frac{n}{2}(n+1) + 3n^2 + \frac{n}{6}(n+1)(2n+1) - \frac{3n}{2}(n+1) + 2n}{2}$$

$$= \frac{n^3 - n^3 - n^2 + 3n^2 + \frac{n}{6}(n+1)(2n+1) - \frac{3n}{2}(n+1) + 2n}{2}$$

H

$$\sum_{i=1}^{n}\frac{(n-i+1)(n-i+2)}{2} = \frac{n^3 - n^3 - n^2 + 3n^2 + \dfrac{n}{6}(n+1)(2n+1) - \dfrac{3n}{2}(n+1) + 2n}{2}$$

$$= \frac{2n^2 + \dfrac{1}{6}(n^2+n)(2n+1) - \dfrac{9}{6}n(n+1) + 2n}{2}$$

$$= \frac{12n^2 + (n^2+n)(2n+1) - 9n^2 - 9n + 12n}{12}$$

$$= \frac{12n^2 + 2n^3 + 3n^2 + n - 9n^2 - 9n + 12n}{12}$$

$$= \frac{2n^3 + 6n^2 + 4n}{12}$$

$$= \frac{n^3 + 3n^2 + 2n}{6}$$

Example : Element uniqueness problem

ALGORITHM   *UniqueElements*(A[0..n − 1])

//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns "true" if all the elements in A are distinct
//   and "false" otherwise
for i ← 0 **to** n − 2 **do**
    for j ← i + 1 **to** n − 1 **do**
        if A[i] = A[j] **return false**
return true

i	j = i+1, …, n-1
0	1, …, n-1
1	2, …, n-1
n-2	n-1

```
      0   1   2   3   4
A  [   |   |   |   |   ]
```
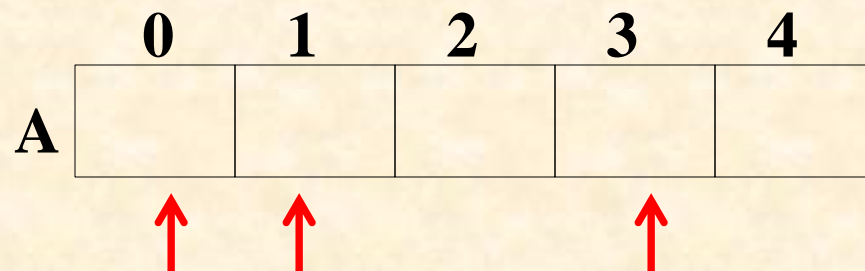
- **The total number of operations is**

$$\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1}1$$

$$\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$\sum_{j=i+1}^{n-1} 1 = (n-1) - (i+1) + 1 = n - i - 1 \qquad \text{\tiny H}$$

H

$$\sum_{i=0}^{n-2} (n-i-1) = (n-1) + (n-2) + \ldots + (1) = \frac{(n-1)(n-2)}{2}$$

- **The running time of the algo. is $O(n^2)$**

Example : Counting binary digits

ALGORITHM $Binary(n)$

//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
$count \leftarrow 1$
while $n > 1$ **do**
    $count \leftarrow count + 1$ **2 3**
    $n \leftarrow \lfloor n/2 \rfloor$ **2 1**
return $count$

2 = 10   3 = 11   4 = 100

n	count
2	2
4	3

Each iteration reduces n by a factor of 2

After i iterations => n will be reduced by 2^i

The halving game: Find integer i such that $n/2^i = 1$.
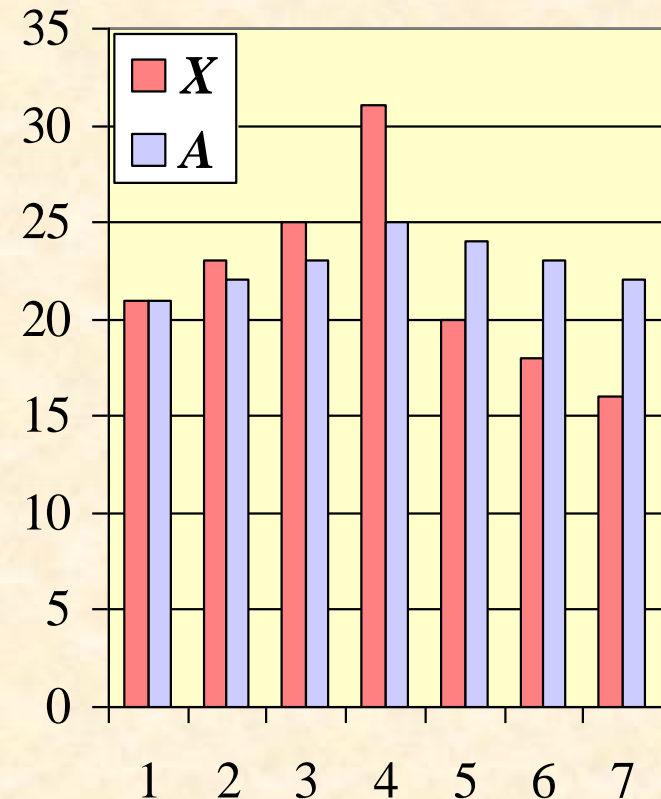
$$\frac{n}{2^i} = 1 \Rightarrow n = 2^i \qquad \Rightarrow i = \lg n$$

- **The running time of the algo. is $O(lgn)$**

Example :Computing Prefix Averages

- We further illustrate asymptotic analysis with two algorithms for prefix averages

- The i-th prefix average of an array X is average of the first $(i + 1)$ elements of X:

    $$A[i] = (X[0] + X[1] + \ldots + X[i])/(i+1)$$

Prefix Averages (Quadratic)

◆ **The following algorithm computes prefix averages in quadratic time by applying the definition**

	#operations
Algorithm *prefixAverages1*(*X, n*)	
Input array *X* **of** *n* **integers**	
Output array *A* **of prefix averages of** *X*	
A ← **new array of** *n* **integers**	*n*
for *i* = **0 to** *n* − **1 do**	*n*
s = *X*[0]	*n*
for *j* = **1 to** *i* **do**	$1 + 2 + \ldots + (n - 1)$
s = *s* + *X*[*j*]	$1 + 2 + \ldots + (n - 1)$
A[*i*] = *s* / (*i* + 1)	*n*
return *A*	1

Arithmetic Progression

- The running time of *prefixAverages1* is $O(1 + 2 + \ldots + n)$

- The sum of the first n integers is $n(n + 1) / 2$
  - There is a simple visual proof of this fact

- Thus, algorithm *prefixAverages1* runs in $O(n^2)$ time
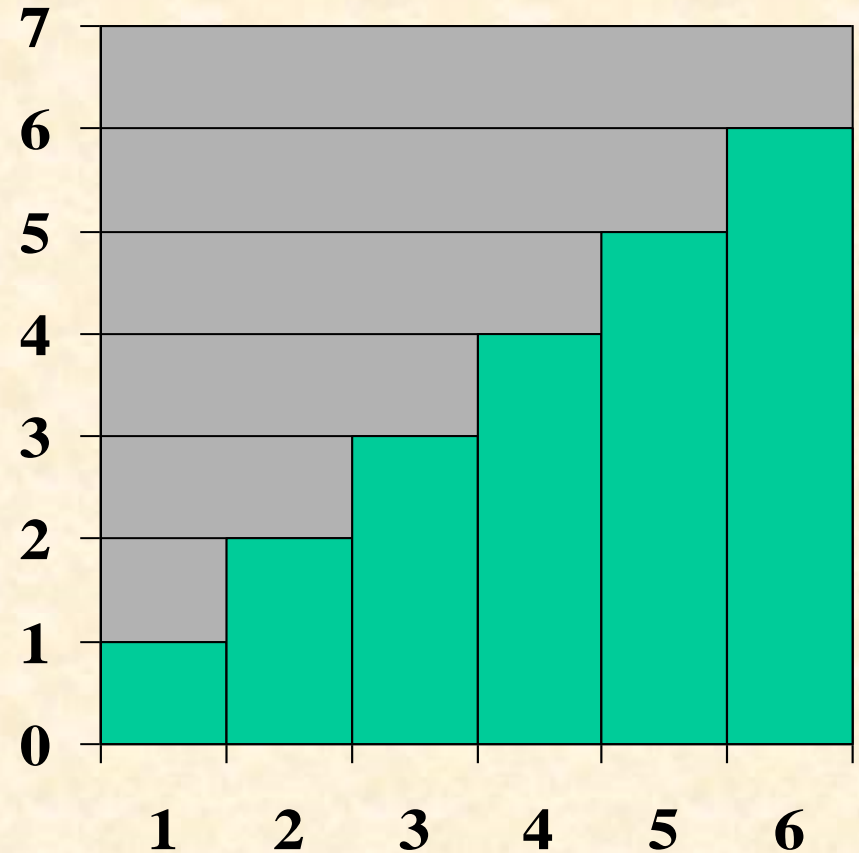
Prefix Averages (Linear)

◆ **The following algorithm computes prefix averages in linear time by keeping a running sum**

	0	1	2
s			

Algorithm *prefixAverages2(X, n)*

  Input array *X* of *n* integers

  Output array *A* of prefix averages of *X*

  A ← **new array of *n* integers**

  s = 0

  for *i* = 0 to *n* − 1 do

    s = *s* + *X[i]*

    A[i] = *s* / (*i* + 1)

  return *A*

i	s	A[i]
0	$X[0]$	$S/1$
1	$X[0] + X[1]$	$S/2$
2	$X[0] + X[1] + X[2]$	$S/3$

◆ **Algorithm *prefixAverages2* runs in $O(n)$ time**

Prefix Averages (Linear)

◆ **The following algorithm computes prefix averages in linear time by keeping a running sum**

	#operations
Algorithm *prefixAverages2(X, n)*	
Input array X of n integers	
Output array A of prefix averages of X	
$A \leftarrow$ **new array of n integers**	n
$s = 0$	1
for $i = 0$ **to** $n - 1$ **do**	n
$s = s + X[i]$	n
$A[i] = s / (i + 1)$	n
return A	1

◆ **Algorithm** *prefixAverages2* **runs in $O(n)$ time**

Check your algorithm

- Suppose the running time of a program is $T(n)$, and on the basis of analysis, the worst-case running time of the program is $O(g(n))$.
  - How do you tell from the measurements made that the program behaves as predicted?
  - Following the definition $T(n) \leq kg(n)$ for $n \geq n_0$, compute the ratio $T(n)/g(n)$ for each of value of n in the experiment and observe how the ratio behaves as n increases. Run-Time Analysis Check your algorithm
    - If the ratio diverges, then $g(n)$ is probably too small
    - if the ratio converges to zero, then $g(n)$ is probably too big
    - if the ratio converges to a constant, then the analysis is probably correct.

Run-Time Analysis

- Running time of algorithm T(n)
- Asymptotic Analysis: $O(g(n)) \Rightarrow T(n) \leq kg(n)$ for $n \geq n_0$
- ratio $T(n)/(g(n))$ diverges
  - as n increases, T(n) increases faster than g(n)
  - Reasons :
    - g(n) is too small; error in analysis of algorithm
- ratio $T(n)/(g(n))$ converges to zero
  - as n increases, g(n) increase faster than T(n)
    - g(n) is too big, i.e. upper bound on time complexity is not tight
    - Errors in analysis of algorithm
    - Worst-case did not arise in experiment
- ratio $T(n)/g(n))$ converges to a constant
  - As n increases, both T(n) & g(n) increase at similar rate
    - Analysis of algorithm is probably correct

Algorithm Analysis

- A problem that has *worst-case polynomial-time*, $O(n^k)$, algorithm is considered "***efficient***"
  - The algorithm can be implement and run for a reasonably large input
  - Such problems are called ***feasible*** or ***tractable***
  - Otherwise, the problem is said to be ***intractable***
    - Any algorithm, if there is one, will take a long time to execute in the worst case, even for modest sizes of input

Unsolvable and NP-Complete

- Some problems are so hard that they have no algorithms at all
    - *Unsolvable* problems

- A large number of solvable problems have an as yet undetermined status
    - They are thought to be intractable but have not been proved to be intractable
    - They are called *NP-complete* problems

Feasible, tractable **P** **NP-complete** **Intractable**

Unsolvable