2.5 Cardinality of Sets

Wenjing Li

wjli@bupt.edu.cn

School of Computer Science

Beijing University of Posts & Telecommunications

Cardinality of Sets

- **Definition:**
  - The number of distinct elements in set *A*, denoted |*A*|, is called the *cardinality* of *A*.
- **Cantor's Definition (1874):**
  - Two sets are defined to have the same cardinality (相同基数) if and only if they can be placed into *one-to-one correspondence* (bijection), and we write |A|=|B|.
- **Note:**
  - Cantor's definition only requires that *some* mapping between the two sets is onto, *not all* are onto.
  - This distinction never arises when the sets are *finite*.

Cardinality of Sets

- **Example:**
  - Do N and E have the same cardinality?
    - N={0,1,2,3,4,5,6,7,…..}
    - E={0,2,4,6,8,10,12,…..} (The even natural numbers.)

- E and N do not have the same cardinality.
- Because E is a proper subset of N with plenty left over.
- The attempted correspondence $f(x)=x$ does not take E onto N.

- There is a bijection f from N to E, the nonnegative even integers, defined by $f(x)=2x$
- The set of even integers has the same cardinality as the set of natural numbers.

Cardinality of Sets

- **More Formal Definition:**
  - For any two (possibly infinite) sets *A* and *B*, we say that *A* and *B* *have the same cardinality* (written $|A|=|B|$) iff there **exists** a bijective function from *A* to *B*.
  - When *A* and *B* are finite, it is easy to see that such a function exists iff *A* and *B* have the same number of elements $n \in N$.
  - When *A* and *B* are infinite, we need to construct such a function to proof the two sets have same cardinality.

N and E have the same cardinality because **there is a** bijective function from N to E, defined by $f(x)=2x$.

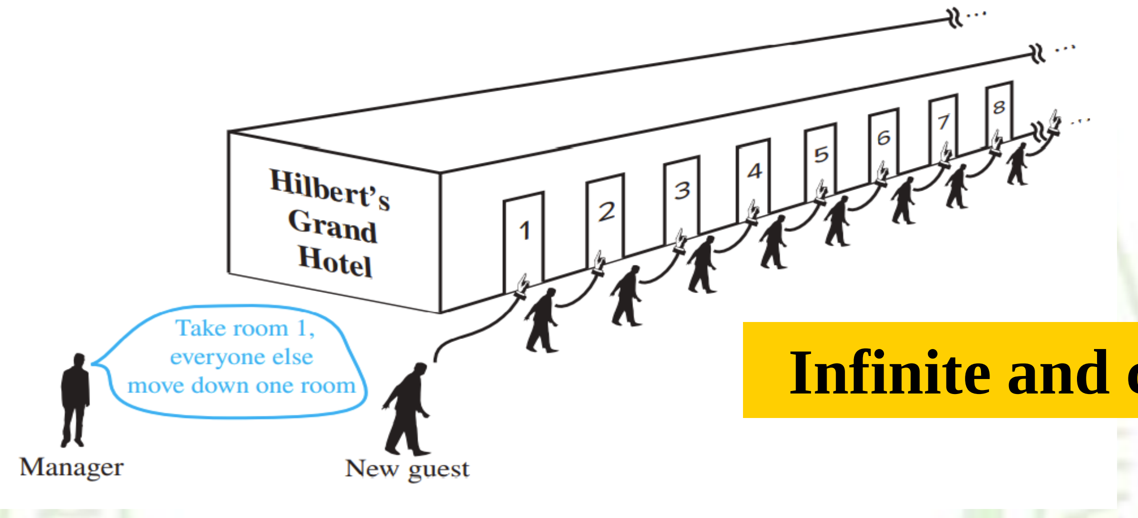
Countable versus Uncountable

- **Definition:**
  - For any set *S*, if *S* is finite or if |S|=|N|, we say that *S* is ***countable***. Else, *S* is ***uncountable***.
  - Intuition behind "**Countable**", we can *enumerate* (sequentially list) elements of *S* in such a way that any individual element of *S* will eventually be counted in the enumeration.
    - Examples: N， Z
  - **Uncountable** means: No series of elements of *S* (even an infinite series) can include all of S's elements.
    - Examples: R, R^2, P(N)

We now split infinite sets into two groups, those with the same cardinality as the set of N and those with a different cardinality.

COUNTABLE SETS

- **Example 2: Hilbert's Grand Hotel**



Infinite and countable

- **Example 3:**
  - **Theorem:** The set Z is countable.
  - **Proof:** Consider $f: Z \rightarrow N$ where $f(i) = \begin{cases} 2i & \text{for } i \geq 0 \\ -2i-1 & \text{for } i < 0 \end{cases}$

    Note f is bijective.

Countable Sets

- **Examples:**
  - **Theorem:** The set of all ordered pairs of natural numbers *(n,m)* is countable.
  - **Proof:** consider listing the pairs in order by their sum *s=n+m*, then by *n*. Every pair appears once in this series; the generating function is bijective.

(0,0)	**(0,1)**	**(0,2)**	**(0,3)**	**(0,4)**	**(0,5)**	**(0,6)**	**……**
(1,0)	**(1,1)**	**(1,2)**	**(1,3)**	**(1,4)**	**(1,5)**	**(1,6)**	**……**
(2,0)	**(2,1)**	**(2,2)**	**(2,3)**	**(2,4)**	**(2,5)**	**(2,6)**	**……**
(3,0)	**(3,1)**	**(3,2)**	**(3,3)**	**(3,4)**	**(3,5)**	**(3,6)**	**……**
(4,0)	**(4,1)**	**(4,2)**	**(4,3)**	**(4,4)**	**(4,5)**	**(4,6)**	**……**
(5,0)	**(5,1)**	**(5,2)**	**(5,3)**	**(5,4)**	**(5,5)**	**(5,6)**	**……**
……	**……**	**……**	**……**				

Uncountable Sets

- **Examples:**
  - **Theorem:** The open interval of reals $[0,1) :\equiv \{ r \in R \mid 0 \le r < 1 \}$ is uncountable.
  - **Proof:** by *diagonalization* (对角线法，Cantor, 1891)
    - Assume there is a series $\{r_i\} = r_1, r_2, \ldots$. Containing all elements $r \in [0,1)$.
    - Consider listing the elements of $\{r_i\}$ in decimal notation (although any base will do) in order of increasing index: …

Georg Cantor
1845-1918

Uncountable Sets

- **Proof (Cont)**

A postulated enumeration of the reals:

$r_1 = 0.d_{1,1}\, d_{1,2}\, d_{1,3}\, d_{1,4}\, d_{1,5}\, d_{1,6}\, d_{1,7}\, d_{1,8}\cdots$

$r_2 = 0.d_{2,1}\, d_{2,2}\, d_{2,3}\, d_{2,4}\, d_{2,5}\, d_{2,6}\, d_{2,7}\, d_{2,8}\cdots$

$r_3 = 0.d_{3,1}\, d_{3,2}\, d_{3,3}\, d_{3,4}\, d_{3,5}\, d_{3,6}\, d_{3,7}\, d_{3,8}\cdots$

$r_4 = 0.d_{4,1}\, d_{4,2}\, d_{4,3}\, d_{4,4}\, d_{4,5}\, d_{4,6}\, d_{4,7}\, d_{4,8}\cdots$

$\cdots$

$r_n = 0.d_{n,1}\, d_{n,2}\, d_{n,3}\, d_{n,4} \ldots\ldots d_{n,n} \ldots\ldots$

Now, consider a real number generated by taking all the digits $d_{i,i}$ that lie along the *diagonal* in this figure and replacing them with *different* digits.

Uncountable Sets

- **Example**
  - A postulated enumeration of the reals:
    $r_1 = 0.301948571\ldots$
    $r_2 = 0.103918481\ldots$
    $r_3 = 0.039194193\ldots$
    $r_4 = 0.918437461\ldots$
  - OK, now let's add 1 to each of the diagonal digits and then mod 10, that is changing 9's to 0.
  - 0.4105… can't be on the list anywhere!

This really doesn't exist in the set.

Transfinite Cardinal Numbers

- **Definition**
  - The cardinalities of infinite sets are not natural numbers, but are special objects called ***transfinite cardinal numbers (超限基数)***.
  - The cardinality of the natural numbers, $\aleph_0 :\equiv |N|$, is the first transfinite cardinal number. (There are none smaller.)

ℵ: *the first letter of the Hebrew alphabet.*

  - The **continuum hypothesis** (连续统假设) claims that $\aleph_1 :\equiv |R|$, the second transfinite cardinal.

Proven impossible to prove or disprove!

Review:Cardinality of sets

- **You should know:**
  - How to define "***same cardinality***" in the case of finite sets and infinite sets.
  - The definitions of ***countable*** and ***uncountable***.
  - How to prove (at least in easy cases) that sets are either countable or uncountable.

- **You should understand:**
  - A finite set must be countable.
  - Infinite sets may be countable, such as N, Z…..
  - Infinite sets may be uncountable, such as R.
  - Transfinite cardinal number： $\aleph_0$, $\aleph_1$

Discrete Mathematics and Its Applications

2.6 Matrices

Wenjing Li

wjli@bupt.edu.cn

School of Computer Science

Beijing University of Posts & Telecommunications

Matrices

- **Definition:**
  - A ***matrix*** is a rectangular array of objects (usually numbers).
  - An $m \times n$ ("m by n") matrix has exactly m horizontal rows, and n vertical columns.
  - Plural of matrix : *matrices*
  - An $n \times n$ matrix is called a *square* matrix, whose *order* or *rank* is n.

$$\begin{bmatrix} 2 & 3 \\ 5 & -1 \\ 7 & 0 \end{bmatrix}$$

a 3×2 matrix

Matrices

- **Notation:**
  - The rows in a matrix are usually indexed 1 to m from top to bottom.
  - The columns are usually indexed 1 to n from left to right.
  - Elements are indexed by row, then column.

$$\mathbf{A} = [a_{i,j}] = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix}$$

Matrices

- **Definition as Functions**
  - An M × N matrix $A=[a_{ij}]$ of member of a set S can be encoded as a ***partial function***

$$f_{\mathbf{A}}: \mathbb{N} \times \mathbb{N} \rightarrow S,$$

such that for $i \leq m, j \leq n, f_A(i,j)=a_{ij}$.

  - By extending the domain over which f_A is defined, various types of infinite and/or multidimensional matrices can be obtained.

Applications of Matrices

- **Tons of applications, including:**
  - Solving systems of linear equations
  - Computer Graphics, Image Processing
  - Models within many areas of Computational Science & Engineering
  - Quantum Mechanics, Quantum Computing
  - Many, many more…

Properties of Matrices

Matrix Equality	矩阵相等
Matrix Sums	矩阵和
Matrix Products	矩阵积
Identity Matrices	单位矩阵
Matrix Inverses	矩阵的逆
Powers of Matrices	矩阵的幂
Matrix Transposition	转置矩阵
Symmetric Matrices	对称矩阵
Zero-One Matrices	0-1矩阵

Matrix Equality

- **Definition:**
  - Two matrices **A** and **B** are considered equal *iff* they have the same number of rows, the same number of columns, and all their corresponding elements are equal.

$$\begin{bmatrix} 3 & 2 \\ -1 & 6 \end{bmatrix} \neq \begin{bmatrix} 3 & 2 & 0 \\ -1 & 6 & 0 \end{bmatrix}$$

Matrix Sums

- **Definition:**
  - The *sum* **A+B** of two matrices **A**, **B** (which **must** have the same number of rows, and the same number of columns) is the matrix (also with the same shape) given by adding corresponding elements of **A** and **B**.

$$\mathbf{A}+\mathbf{B} = [a_{i,j}+b_{i,j}]$$

$$\begin{bmatrix} 2 & 6 \\ 0 & -8 \end{bmatrix} + \begin{bmatrix} 9 & 3 \\ -11 & 3 \end{bmatrix} = \begin{bmatrix} 11 & 9 \\ -11 & -5 \end{bmatrix}$$

Matrix Products

- **Definition:**
  - For an $m \times k$ matrix **A** and a $k \times n$ matrix **B**, the *product* **AB** is the $m \times n$ matrix:

$$\mathbf{AB} = \mathbf{C} = [c_{i,j}] \equiv \left[\sum_{\ell=1}^{k} a_{i,\ell} b_{\ell,j} \right]$$

    - The element of **AB** indexed (i,j) is given by the ***vector dot product*** **(向量点积)** of the <u>ith row of **A**</u> and the <u>jth column of **B**</u> (considered as vectors).
    - **Note:** Matrix multiplication is <u>not</u> commutative!
- **Example:**

$$\begin{bmatrix} 0 & 1 & -1 \\ 2 & 0 & 3 \end{bmatrix} \cdot \begin{bmatrix} 0 & -1 & 1 & 0 \\ 2 & 0 & -2 & 0 \\ 1 & 0 & 3 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -5 & -1 \\ 3 & -2 & 11 & 3 \end{bmatrix}$$

Matrix Products

- **Matrix Multiplication Algorithm:**

procedure *matmul*(matrices **A**: $m \times k$, **B**: $k \times n$)

for $i := 1$ **to** m (m)

  for $j := 1$ **to** n **begin** (n)

    $c_{ij} := 0$ (1)

    for $q := 1$ **to** k (k)

      $c_{ij} := c_{ij} + a_{iq}b_{qj}$ (1)

end

m*(n*(1+k*1))

{**C**=[c_{ij}] is the product of **A** and **B**}

What's the Θ of its time complexity? **Answer: Θ(m·n·k)**

Identity Matrices

- **Definition:**
  - The *identity matrix of order n*, $\mathbf{I}_n$, is the rank-*n* square matrix with 1's along the upper-left to lower-right diagonal, and 0's everywhere else.

$\forall 1 \le i,j \le n$

$$\mathbf{I}_n = [\delta_{ij}] = \left[\begin{cases} 1 \text{ if } i = j \\ 0 \text{ if } i \ne j \end{cases}\right] = \underbrace{\begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}}_{n} \left.\vphantom{\begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}}\right\} n$$

Kronecker Delta

克罗内克 δ 函数

最深奥的数学研究的结果，最终都一定可以表示成整数性质的简单形式

Matrix Inverses

- **Definition:**
  - For some (but not all) square matrices $\mathbf{A}$, there exists a unique multiplicative *inverse* $\mathbf{A}^{-1}$ of $\mathbf{A}$, a matrix such that $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}_n$.
  - If the inverse exists, it is unique, and $\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1}$.
  - We won't go into the algorithms for matrix inversion...

How to get the matrix inverse?

Powers of Matrices

- **Definition:**

  If **A** is an $n \times n$ square matrix and $p \geq 0$, then:

  - $\mathbf{A}^p :\equiv \underbrace{\mathbf{AAA}\cdots\mathbf{A}}_{\text{p times}}$ (and $\mathbf{A}^0 :\equiv \mathbf{I}_n$)

- **Example:**

$$\begin{bmatrix} 2 & 1 \\ -1 & 0 \end{bmatrix}^3 = \begin{bmatrix} 2 & 1 \\ -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 2 & 1 \\ -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 2 & 1 \\ -1 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 2 & 1 \\ -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 3 & 2 \\ -2 & -1 \end{bmatrix}$$

$$= \begin{bmatrix} 4 & 3 \\ -3 & -2 \end{bmatrix}$$

Matrix Transposition

- **Definition:**
  - If $\mathbf{A}=[a_{ij}]$ is an $m \times n$ matrix, the *transpose* of $\mathbf{A}$ (often written $\mathbf{A}^t$ or $\mathbf{A}^T$) is the $n \times m$ matrix given by $\mathbf{A}^t = \mathbf{B} = [b_{ij}] = [a_{ji}]$ $(1 \le i \le n, 1 \le j \le m)$

$$\begin{bmatrix} 2 & 1 & 3 \\ 0 & -1 & -2 \end{bmatrix}^t = \begin{bmatrix} 2 & 0 \\ 1 & -1 \\ 3 & -2 \end{bmatrix}$$

Flip across diagonal

$$\begin{bmatrix} 2 & 3 & -2 \\ 4 & -3 & 1 \\ -1 & 2 & 5 \end{bmatrix}^t = \begin{bmatrix} 2 & 4 & -1 \\ 3 & -3 & 2 \\ -2 & 1 & 5 \end{bmatrix}$$

Symmetric Matrices

- **Definition:**
  - A square matrix **A** is *symmetric* iff $\mathbf{A}=\mathbf{A}^t$.
  - *I.e.*, $\forall i,j \le n$: $a_{ij} = a_{ji}$.
- **Example:**
  - Which of the below matrices is symmetric?

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \qquad \begin{bmatrix} -2 & 1 & 3 \\ 1 & 0 & -1 \\ 3 & -1 & 2 \end{bmatrix} \qquad \begin{bmatrix} 3 & 0 & 1 \\ 0 & 2 & -1 \\ 1 & 1 & -2 \end{bmatrix}$$

Zero-One Matrices

- **Definition:**
  - All elements of a *zero-one* matrix are either 0 or 1, Representing **False** & **True** respectively.
  - Useful for representing other structures. *E.g.*, relations, directed graphs (later in this course).
- **Operation:**
  - The *join (并)* of **A**, **B** (both $m \times n$ zero-one matrices):
  - $\mathbf{A} \vee \mathbf{B} :\equiv [a_{ij} \vee b_{ij}]$
  - The *meet (交)* of **A**, **B**:
  - $\mathbf{A} \wedge \mathbf{B} :\equiv [a_{ij} \wedge b_{ij}] = [a_{ij}\, b_{ij}]$

The 1's in A join the 1's in B to make up the 1's in C.

Where the 1's in A meet the 1's in B, we find 1's in C.

Zero-One Matrices

- **Example:**
  - Find the join and meet of the zero-one matrices

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix}.$$

  - **Solution:** The join of A and B is

$$\mathbf{A} \vee \mathbf{B} = \begin{bmatrix} 1 \vee 0 & 0 \vee 1 & 1 \vee 0 \\ 0 \vee 1 & 1 \vee 1 & 0 \vee 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix}.$$

  - **Solution:** The meet of A and B is

$$\mathbf{A} \wedge \mathbf{B} = \begin{bmatrix} 1 \wedge 0 & 0 \wedge 1 & 1 \wedge 0 \\ 0 \wedge 1 & 1 \wedge 1 & 0 \wedge 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

0-1 Matrices: Boolean Products

- **Definition:**
  - Let **A**=[a_{ij}] be an $m \times k$ zero-one matrix, & let **B**=[b_{ij}] be a $k \times n$ zero-one matrix,
  - The *boolean product* (布尔积) of **A** and **B** is like normal matrix ×, but using ∨ instead of + in the row-column "***vector dot product***":

$$\mathrm{A} \odot \mathrm{B} = \mathbf{C} = [c_{ij}] = \left[\bigvee_{\ell=1}^{k} a_{i\ell} \wedge b_{\ell j} \right]$$

$$c_{ij} = (a_{i1} \wedge b_{1j}) \vee (a_{i2} \wedge b_{2j}) \vee \ldots \vee (a_{ik} \wedge b_{kj})$$

0-1 Matrices: Boolean Product

- **Example:**
  - Find the Boolean product of A and B, where

$$\mathbf{A} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}.$$

  - **Solution**: The Boolean product of $\mathbf{A} \odot \mathbf{B}$ is?

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} (1 \wedge 1) \vee (0 \wedge 0) & (1 \wedge 1) \vee (0 \wedge 1) & (1 \wedge 0) \vee (0 \wedge 1) \\ (0 \wedge 1) \vee (1 \wedge 0) & (0 \wedge 1) \vee (1 \wedge 1) & (0 \wedge 0) \vee (1 \wedge 1) \\ (1 \wedge 1) \vee (0 \wedge 0) & (1 \wedge 1) \vee (0 \wedge 1) & (1 \wedge 0) \vee (0 \wedge 1) \end{bmatrix}$$

$$= \begin{bmatrix} 1 \vee 0 & 1 \vee 0 & 0 \vee 0 \\ 0 \vee 0 & 0 \vee 1 & 0 \vee 1 \\ 1 \vee 0 & 1 \vee 0 & 0 \vee 0 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix}.$$

0-1 Matrices: Boolean Powers

- **Definition:**
  - For a square zero-one matrix **A**, and any $k \geq 0$, the k_{th} *Boolean power of* **A** is simply the Boolean product of k copies of **A**.

$$\mathbf{A}^{[k]} :\equiv \underbrace{\mathbf{A} \odot \mathbf{A} \odot \ldots \odot \mathbf{A}}_{\text{k times}}$$

$$\mathbf{A}^{[0]} :\equiv I_n$$

0-1 Matrices: Boolean Powers

- **Example:**
  - Let $\mathbf{A} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix}$. Find $\mathbf{A}^{[n]}$ for all positive integers n.

- **Solution:**

$$\mathbf{A}^{[2]} = \mathbf{A} \odot \mathbf{A} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

$$\mathbf{A}^{[3]} = \mathbf{A}^{[2]} \odot \mathbf{A} = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

$$\mathbf{A}^{[4]} = \mathbf{A}^{[3]} \odot \mathbf{A} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$$\mathbf{A}^{[5]} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$\mathbf{A}^{[\mathbf{n}]} = \mathbf{A}^{\mathbf{5}}$ for all positive integers n with $n \geq 5$.

0-1 Matrices: Basic properties

- If $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ are Boolean matrices of compatible sizes, then
  - $A \vee B = B \vee A$ Commutative law
  - $A \wedge B = B \wedge A$
  - $(A \vee B) \vee C = A \vee (B \vee C)$ Associative law
  - $(A \wedge B) \wedge C = A \wedge (B \wedge C)$
  - $A \wedge (B \vee C) = (A \vee B) \wedge (A \vee C)$ Distributive law
  - $A \vee (B \wedge C) = (A \wedge B) \vee (A \wedge C)$
  - $(A \odot B) \odot C = A \odot (B \odot C)$

Review: Matrices

- **We have learned:**
  - Definition and notation of matrix.
  - Arithmetic of matrices:
    - Metrix Sum
    - Metrix Product
  - Some special matrices:
    - Indentity Matrix
    - Matrix Inverses
    - Powers of Matrices
    - Matrix Transposition
    - Symmetric Matrices
    - Zero-one Matrix and its properties

Homework

- **§ 2.5**
  - 2, 10
- **§ 2.6**
  - 4(b), 20, 28, 32

3.1 Algorithms

Wenjing Li

wjli@bupt.edu.cn

School of Computer Science

Beijing University of Posts & Telecommunications

Algorithms

- The foundation of computer programming.
- Most generally, an *algorithm* just means a definite procedure for performing some sort of task.
- A computer *program* is simply a description of an algorithm, in a language precise enough for a computer to understand, requiring only operations that the computer already knows how to do.
- We say that a program *implements* (or "is an implementation of") its algorithm.

Algorithms You Already Know

- **Grade-school arithmetic algorithms:**
  - How to add any two natural numbers written in decimal on paper, using carries.
  - Similar: Subtraction using borrowing.
  - Multiplication & long division.

Executing an Algorithm

- When you start up a piece of software, we say the program or its algorithm are being *run* or *executed* by the computer.
- Given a description of an algorithm, you can also execute it by hand, by working through all of its steps with pencil & paper.
- Before ~1940, “computer” meant a *person* whose job was to execute algorithms!

We will learn

- An informal "*pseudo-code*" language.
- Some *basic algorithms*:
  - Max algorithm
  - Primality-testing
  - Searching: linear search & binary search
  - Sorting: bubble sort & insertion sort
  - Greedy

Algorithm Example: Max

- **Task:**
  - Given a sequence $\{a_i\}=a_1,\ldots,a_n$, $a_i\in\mathbf{N}$, say what its largest element is.
  - One algorithm for doing this:
    - Set the value of a *temporary variable v* (largest element seen so far) to a_1.
    - Look at the next element a_i in the sequence.
    - If $a_i>v$, then re-assign v to the number a_i.
    - Repeat previous 2 steps until there are no more elements in the sequence, & return v.

Algorithm Example: Max

- **Running the Algorithm**
  - Let $\{a_i\}$=7, 12, 3, 15, 8. Find its maximum…
    - Set $v = a_1 = 7$.
    - Look at next element: $a_2 = 12$.
    - Is $a_2>v$? Yes, so change v to 12.
    - Look at next element: $a_2 = 3$.
    - Is 3>12? No, leave v alone….
    - Is 15>12? Yes, v=15…

Algorithm Characteristics

- **Some important features of algorithm:**
  - *Input(输入).* Information or data that comes in.
  - *Output (输出).* Information or data that goes out.
  - *Definiteness(确定性).* Algorithm is precisely defined.
  - *Correctness(正确性).* Outputs correctly relate to inputs.
  - *Finiteness(有限性).* Won't take forever to describe or run.
  - *Effectiveness(有效性).* Individual steps are all do-able.
  - *Generality(通用性).* Works for many possible inputs.
  - *Efficiency(高效性).* Takes little time & memory to run.

Programming Languages

- Some common programming languages:
  - **Older:** Fortran, Cobol, Lisp, Pascal, Basic
  - **Newer:** Java, C, C++, C#, Visual Basic, JavaScript, Perl, Tcl, Python, many others…
  - **Assembly languages:** for low-level coding.
- In this class we will use an informal, Pascal-like "***pseudo-code***" language.
- You should know at least 1 real language!

Pseudocode Language

procedure
<u>*procname(argument: type)*</u>
<u>*variable*</u> := <u>*expression*</u>
<u>*statement*</u>
<u>*informal statement*</u>
return <u>*expression*</u>

begin <u>*statements*</u> **end**

{<u>*comment*</u>**}**

if <u>*condition*</u> then <u>*statement*</u>
[else <u>*statement*</u>]

for <u>*variable*</u> := <u>*initial value*</u>
to <u>*final value*</u>
<u>*statement*</u>

while <u>*condition*</u>
<u>*statement*</u>

PROCEDURE <u>*procname*</u> (<u>*arg*</u>: <u>*type*</u>)

- Declares that the following text defines a procedure named <u>*procname*</u> that takes inputs (*arguments*) named <u>*arg*</u> which are data objects of the type <u>*type*</u>.
  - **Example:**
    procedure *maximum*(*L*: list of integers)
      [statements defining *maximum*…]
    return *expression*

- Various real programming languages refer to procedures as *functions* (since the procedure call notation works similarly to function application *f*(*x*)), or as *subroutines*, *subprograms*, or *methods*.

variable := *expression*

- An *assignment* statement evaluates the expression *expression*, then reassigns the variable *variable* to the value that results.
  - **Example**

    assignment statement:

    $v := 3x+7$ (If x is 2, changes v to 13.)

- In pseudocode (but not real code), the *expression* might be informally stated:
  - $x :=$ the largest integer in the list L

Informal statement

- Sometimes we may write a statement as an informal English imperative, if the meaning is still clear and precise: *e.g.*,“swap *x* and *y*”

- Keep in mind that real programming languages never allow this.

begin *statements* end

- Groups a sequence of statements together:

```
begin
    statement 1
    statement 2
    ...
    statement n
end
```

Curly braces {} are used instead in many languages.

- Might be used:
  - After a **procedure** declaration.
  - In an **if** statement after **then** or **else**.
  - In the body of a **for** or **while** loop.

{comment}

- Not executed (does nothing).
- Natural-language text explaining some aspect of the procedure to human readers.
- Also called a *remark* in some real programming languages, *e.g.* BASIC.
- **Example**
  - Might appear in a *max* program: {Note that *v* is the largest integer seen so far.}

if *condition* then *statement*

- Evaluate the propositional expression *condition*.
  - If the resulting truth value is **True**, then execute the statement *statement*;
  - otherwise, just skip on ahead to the next statement after the **if** statement.

- **if** *cond* **then** *stmt1* **else** *stmt2*
  - Like above, but iff truth value is **False**, executes *stmt2*.

while *condition statement*

- *Evaluate* the propositional (Boolean) expression *condition*. If the resulting value is **True**, then execute *statement*.
- Continue repeating the above two actions over and over until finally the *condition* evaluates to **False**; then proceed to the next statement.
- Equivalent to infinite nested **if**s:

if *condition*
    begin
        statement
        if *condition*
            begin
                statement
                …(infinite nested if's)
            end
    end

for *var* := *initial* to *final* *stmt*

- *Initial* is an integer expression.
- *Final* is another integer expression.
- **Semantics:**
  - Repeatedly execute *stmt*, first with variable *var* := *initial*, then with *var* := *initial*+1, then with *var* := *initial*+2, *etc*., then finally with *var* := *final*.
- **Question:**
  - What happens if *stmt* changes the value of var, or the value that *initial* or *final* evaluates to?

for *var* := *initial* to *final stmt*

- **For** can be exactly defined in terms of **while,** like so:

```
begin
    var := initial
    while var ≤ final
        begin
            stmt
            var := var + 1
        end
end
```

Max procedure in pseudocode

procedure *max* ($a_1, a_2, \ldots, a_n$: integers)

    $v := a_1$ {largest element so far}

    for $i := 2$ **to** n {go thru rest of elems}

        if $a_i > v$ **then** $v := a_i$ {found bigger?}

    {at this point v's value is the same as the largest integer in the list}

    return v

Input	Finiteness
Output	Effectiveness
Definiteness	Generality
Correctness	Efficiency

Inventing an Algorithm

- Requires a lot of *creativity and intuition*
  - Like writing proofs.

- Unfortunately, we can't give you an algorithm for inventing algorithms.
  - Just look at lots of examples…
  - And practice (preferably, on a computer)
  - And look at more examples…
  - And practice some more… etc., etc.

Algorithm Example: *IsPrime*

- Suppose we ask you to write an algorithm to compute the predicate:

$$IsPrime: \mathbf{N} \rightarrow \{T,F\}$$

  - Computes whether a given natural number is a prime number.

- First, start with a correct predicate-logic definition of the desired function:

$$\forall n: IsPrime(n) \Leftrightarrow \neg\exists 1<d<n: \underbrace{d|n}$$

Means d divides n evenly (without remainder)

Algorithm Example: *IsPrime*

- Notice that the negated existential can be rewritten as a universal:

$$\neg\exists 1<d<n\colon d|n \Leftrightarrow \quad \forall 1<d<n\colon\ \neg d \mid n$$

$$\Leftrightarrow \quad \forall 2\leq d \leq n-1\colon\ \neg d \mid n$$

- This universal can then be translated directly into a corresponding **for** loop:

for *d* := 2 to *n*−1 { Try all potential divisors >1 & <n }
    if *d*|*n* **then return** F { n has divisor d; not prime }
return T { no divisors were found; n must be prime}

Iteration number: ***n-2***

Algorithm Example: *IsPrime*

- The *IsPrime* algorithm can be ***further optimized***:

```
for d := 2 to ⌊n^(1/2)⌋
    if d|n then return F
return T
```

$$\textbf{for } d := 2 \text{ to } \lfloor n^{1/2} \rfloor$$
$$\quad \textbf{if } d|n \textbf{ then return F}$$
$$\textbf{return T}$$

only try divisors that are primes less than $n^{1/2}$.

- This works because of this **theorem**:
  - If n has any (integer) divisors, it must have one less than $n^{1/2}$.
  - **Proof:** Suppose n's smallest divisor >1 is a, and let b :≡ n/a, then n = ab. If $a > n^{1/2}$ then $b > n^{1/2}$ (since a is n's smallest divisor) and so $n = ab > (n^{1/2})^2 = n$, an absurdity.

Algorithm Example: *Searching*

- Problem of ***searching*** *an ordered list*.
  - Given a list L of n elements that are sorted into a definite order (*e.g.*, numeric, alphabetical),
  - And given a particular element x,
  - Determine whether x appears in the list,
  - and if so, return its index (position) in the list.
- Problem occurs often in many contexts
  - E.g. Database, Library, Web…
- Let's find an *efficient* algorithm!

Search alg. #1: *Linear Search*

procedure *linear search*

(x: integer, $a_1, a_2, \ldots, a_n$: distinct integers)

$i := 1$ {start at beginning of list}

while $(i \le n \wedge x \ne a_i)$ {not done, not found}

$i := i + 1$ {go to the next position}

if $i \le n$ **then** *location* $:= i$ {it was found}

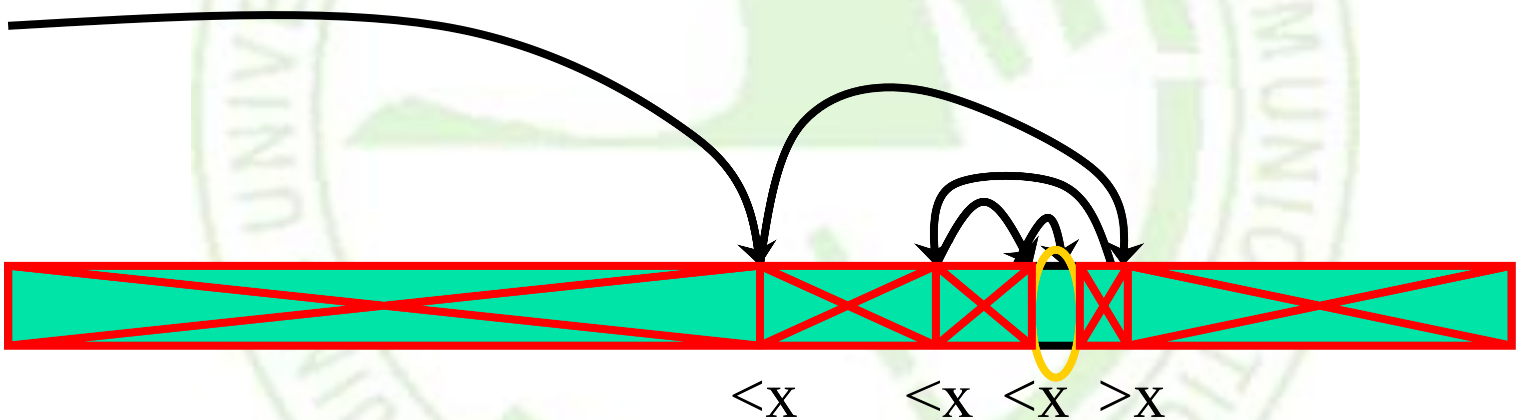
else *location* $:= 0$ {it wasn't found}

return *location* {index or 0 if not found}

Worst iteration number: $\boldsymbol{n}$

Search alg. #2: *Binary Search*

- **Basic idea:**
  - On each step, look at the *middle* element of the remaining list to eliminate half of it, and quickly zero in on the desired element.



Search alg. #2: *Binary Search*

procedure ***binary search***

(x: integer, a_1, a_2, …, a_n: distinct integers)

$i := 1$ {left endpoint of search interval}

$j := n$ {right endpoint of search interval}

while $i<j$ **begin** {while interval has >1 item}

$m := \lfloor (i+j)/2 \rfloor$ {find midpoint}

if $x>a_m$ **then** $i := m+1$

else $j := m$ {eliminate half of it}

end

if $x = a_i$ **then** *location* $:= i$ **else** *location* $:= 0$

return *location*

*Worst iteration number: **?***

Practice exercises

- Devise an algorithm that finds the sum of all the integers in a list.

procedure ***sum***$(a_1, a_2, \ldots, a_n$: *integers*)

    $s := 0$ *{sum of elems so far}*

    for $i := 1$ *to* n *{go thru all elems}*

        $s := s + a_i$ *{add current item}*

        {at this point s is the sum of all items}

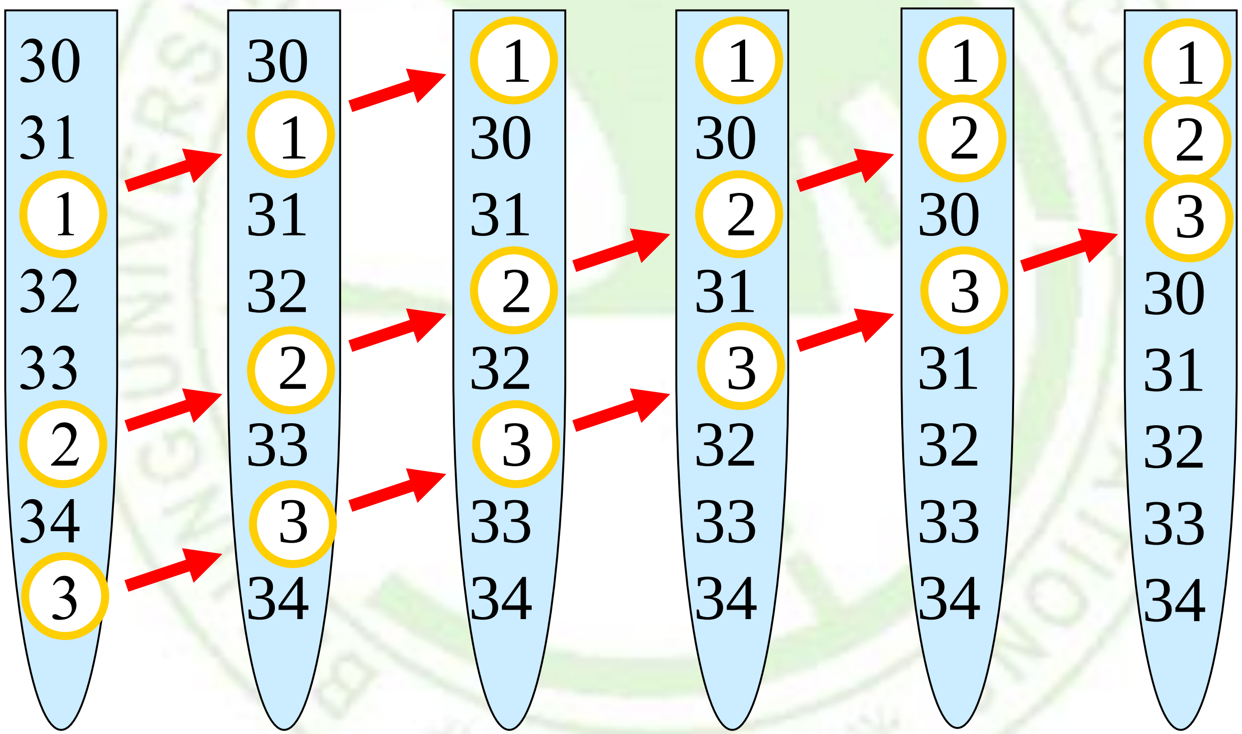
    return *s*

Algorithm Example: *Sorting*

- ***Sorting*** is a common operation in many applications.
  - *E.g.* spreadsheets and databases
- It is also widely used as a subroutine in other data-processing algorithms.
- Two sorting algorithms shown in textbook:
  - Bubble sort
  - Insertion sort

Sorting alg. #1: *Bubble Sort*

- Smallest elements "float" up to the top of the list, like bubbles in a container of liquid.



Worst iteration number: ***?***

Sorting alg. #2: *Insertion Sort*

- For each item in the input list,
  - “Insert” it into the correct place in the sorted output list generated so far. Like so:
    - Use linear or binary search to find the location where the new item should be inserted.
    - Then, shift the items from that position onwards down by one position.
    - Put the new item in the hole remaining.

We’ll see some more efficient sort algorithms later in the course.

Algorithm Example: *Greedy*

- **Example 6**
  - Consider the problem of making n cents change with ***quarters(25), dimes(10), nickels(5),*** and ***pennies(1)***, and using the least total number of coins.
  - We can devise a greedy algorithm for making change for n cents by making a locally optimal choice at each step.

ALGORITHM 6 Greedy Change-Making Algorithm.

procedure *change*($c_1, c_2, \ldots, c_r$: values of denominations of coins, where
  $c_1 > c_2 > \cdots > c_r$; n: a positive integer)
for $i := 1$ **to** r
  $d_i := 0$ {d_i counts the coins of denomination c_i used}
  while $n \geq c_i$
    $d_i := d_i + 1$ {add a coin of denomination c_i }
    $n := n - c_i$
{d_i is the number of coins of denomination c_i in the change for $i = 1, 2, \ldots, r$}

GREEDY EXAMPLE: *LEMMA 1*

- If *n* is a positive integer, then *n* cents in change using ***quarters, dimes, nickels***, and ***pennies*** using the fewest coins possible has:
  - at most two dimes,
  - at most one nickel,
  - at most four pennies, and
  - cannot have two dimes and a nickel,
  - the amount of change in dimes, nickels, and pennies cannot exceed 24 cents.

At most 9 cents

Proof: By contradiction

- If we had 3 dimes, we could replace them with a quarter and a nickel.
- If we had 2 nickels, we could replace them with 1 dime.
- If we had 5 pennies, we could replace them with a nickel.
- If we had 2 dimes and 1 nickel, we could replace them with a quarter.
- The allowable combinations, have a maximum value of 24 cents; 2 dimes and 4 pennies.

Greedy Example: *Theorem 1*

- The greedy algorithm (Algorithm 6) produces change using the *fewest coins* possible.

本例中贪心算法会得到最优解

- **Proof: By contradiction.**
  - Assume there is a positive integer n such that change can be made for n cents using quarters, dimes, nickels, and pennies, with a fewer total number of coins than given by the algorithm.
  - Let q' is the number of quarters used in this optimal way and q is the number of quarters in the greedy algorithm's solution, then, $q' \leq q$ (why?). But $q' < q$ is not possible by Lemma 1, since the value of the coins other than quarters can not be greater than 24 cents.
  - Similarly, by Lemma 1, the two algorithms must have the same number of dimes, nickels, and quarters.

Review: Algorithms

- Characteristics of algorithms.
- Pseudocode.
- Examples:
  - Max alg, primality-testing alg, linear search & binary search alg, bubble & insertion sorting alg, greedy alg.
  - Intuitively we see that binary search is much faster than linear search, but how do we analyze the efficiency of algorithms formally?
  - Use methods of ***algorithmic complexity***, which utilize the order-of-growth concepts from **§ 3.2**.

Homework

- § 3.1
  - 56 (a)(c)