CSE 15
Discrete Mathematics

Lecture 11 – Matrices

Algorithms

Announcement

- HW #5 out
  - Due **5pm** 10/19 (Fri) with 1 extra day of re-submission.
- Reading assignment
  - Ch. 4.1 – 4.3 of textbook

Matrices (Ch. 2.6)

- Definition of a Matrix
- Matrix Arithmetic
- Transposes and Powers of Arithmetic
- Zero-One matrices

Matrices

- Matrices are useful discrete structures that can be used in many ways. For example, they are used to:
  - Describe certain types of functions known as linear transformations.
  - Express which vertices of a graph are connected by edges (see Chapter 10).
- Matrices can be used to build models of:
  - Transportation systems.
  - Communication networks.
- We cover the aspect of matrix arithmetic that will be needed later.

Matrix

Definition: A *matrix* is a rectangular array of numbers. A matrix with *m* rows and *n* columns is called an $m \times n$ matrix.

- The plural of matrix is *matrices*.
- A matrix with the same number of rows as columns is called *square*.
- Two matrices are *equal* if they have the same number of rows and the same number of columns and the corresponding entries in every position are equal.

$$3 \times 2 \text{ matrix} \qquad \begin{bmatrix} 1 & 1 \\ 0 & 2 \\ 1 & 3 \end{bmatrix}$$

Notation

- Let m and n be positive integers and let

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ . & . & & . \\ . & . & & . \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

- The ith row of **A** is the $1 \times n$ matrix $[a_{i1}, a_{i2}, \ldots, a_{in}]$. The jth column of **A** is the $m \times 1$ *matrix:*

$$\begin{bmatrix} a_{1j} \\ a_{2j} \\ . \\ . \\ a_{mj} \end{bmatrix}$$

- The (i,j)th *element* or *entry* of **A** is the element a_{ij}. We can use **A** = $[a_{ij}]$ to denote the matrix with its (i,j)th element equal to a_{ij}.

Matrix Arithmetic: Addition

Defintion: Let $\mathbf{A} = [\mathrm{a}_{\mathrm{ij}}]$ and $\mathbf{B} = [\mathrm{b}_{\mathrm{ij}}]$ be $m \times n$ matrices. The sum of **A** and **B**, denoted by **A + B**, is the $m \times n$ matrix that has $a_{\mathrm{ij}} + b_{\mathrm{ij}}$ as its (i,j)th element. In other words, **A + B =** $[a_{ij} + b_{ij}]$**.**

Example:

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 2 & -3 \\ 3 & 4 & 0 \end{bmatrix} + \begin{bmatrix} 3 & 4 & -1 \\ 1 & -3 & 0 \\ -1 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 4 & 4 & -2 \\ 3 & -1 & -3 \\ 2 & 5 & 2 \end{bmatrix}$$

Note that matrices of different sizes can not be added.

Matrix Multiplication

Definition: Let **A** be an $m \times k$ matrix and **B** be a $k \times n$ matrix. The *product* of **A** and **B**, denoted by **AB**, is the $m \times n$ matrix that has its (*i,j*)th element equal to the sum of the products of the corresponding elements from the *i*th row of **A** and the *j*th column of **B**. In other words, if **AB** = $[c_{ij}]$ then $c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \ldots + a_{ik}b_{kj}$.

Example:

$$\begin{bmatrix} 1 & 0 & 4 \\ 2 & 1 & 1 \\ 3 & 1 & 0 \\ 0 & 2 & 2 \end{bmatrix} \begin{bmatrix} 2 & 4 \\ 1 & 1 \\ 3 & 0 \end{bmatrix} = \begin{bmatrix} 14 & 4 \\ 8 & 9 \\ 7 & 13 \\ 8 & 2 \end{bmatrix}$$

The product of two matrices is undefined when the number of columns in the first matrix is not the same as the number of rows in the second.

Illustration of Matrix Multiplication

- The Product of **A** = [a$_{ij}$] and **B** = [b$_{ij}$]

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1k} \\ a_{21} & a_{22} & \dots & a_{2k} \\ . & . & & . \\ . & . & & . \\ a_{i1} & a_{i2} & \dots & a_{ik} \\ . & . & & . \\ . & . & & . \\ a_{m1} & a_{m2} & \dots & a_{mk} \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} b_{11} & a_{12} & \dots & b_{1j} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2j} & \dots & b_{2n} \\ . & . & & . & & \\ . & . & & . & & \\ b_{k1} & b_{k2} & \dots & b_{kj} & \dots & b_{kn} \end{bmatrix}$$

$$\mathbf{AB} = \begin{bmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ . & . & & . \\ . & . & c_{ij} & . \\ . & . & & . \\ c_{m1} & c_{m2} & \dots & c_{mn} \end{bmatrix}$$

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{ik}b_{kj}$$

Matrix Multiplication is not Commutative

Example: Let

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$$

Does **AB** = **BA**?

Solution:

$$\mathbf{AB} = \begin{bmatrix} 2 & 2 \\ 5 & 3 \end{bmatrix} \quad \mathbf{BA} = \begin{bmatrix} 4 & 3 \\ 3 & 2 \end{bmatrix}$$

AB $\neq$ **BA**

Identity Matrix and Powers of Matrices

Definition: The *identity matrix of order n* is the $n \times n$ matrix $\mathbf{I}_n = [\delta_{ij}]$, where $\delta_{ij} = 1$ if $i = j$ and $\delta_{ij} = 0$ if $i \neq j$.

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

$$\mathbf{AI}_n = \mathbf{I}_m\mathbf{A} = \mathbf{A}$$

when $\mathbf{A}$ is an $m \times n$ matrix

Powers of square matrices can be defined. When A is an $n \times n$ matrix, we have:

$$\mathbf{A}^0 = \mathbf{I}_n \qquad \mathbf{A}^r = \underbrace{\mathbf{AAA}\cdots\mathbf{A}}_{\text{r times}}$$

Transposes of Matrices

Definition: Let $\mathbf{A} = [a_{ij}]$ be an $m \times n$ matrix. The *transpose* of **A**, denoted by $\mathbf{A}^t$,is the $n \times m$ matrix obtained by interchanging the rows and columns of **A**.

If $\mathbf{A}^t = [b_{ij}]$, then $b_{ij} = a_{ji}$ for $i = 1,2,\ldots,n$ and $j = 1,2, \ldots,m$.

The transpose of the matrix $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ is the matrix $\begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$.

Transposes of Matrices

Definition: A square matrix **A** is called ***symmetric*** if **A** = $\mathbf{A}^t$. Thus **A** = $[a_{ij}]$ is symmetric if $a_{ij} = a_{ji}$ for i and j with $1 \leq i \leq n$ and $1 \leq j \leq n$.

Example:

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

Zero-One Matrices

Definition: A matrix all of whose entries are either 0 or 1 is called a *zero-one matrix*.

Algorithms operating on discrete structures represented by zero-one matrices are based on Boolean arithmetic defined by the following Boolean operations:

$$b_1 \wedge b_2 = \begin{cases} 1 & \text{if } b_1 = b_2 = 1 \\ 0 & \text{otherwise} \end{cases} \quad b_1 \vee b_2 = \begin{cases} 1 & \text{if } b_1 = 1 \text{ or } b_2 = 1 \\ 0 & \text{otherwise} \end{cases}$$

Zero-One Matrices

Definition: Let $\mathbf{A} = [a_{ij}]$ and $\mathbf{B} = [b_{ij}]$ be $m \times n$ zero-one matrices.

- The *join* of **A** and **B** is the zero-one matrix with (i,j)th entry $a_{ij} \vee b_{ij}$. The *join* of **A** and **B** is denoted by $\mathbf{A} \vee \mathbf{B}$.
- The *meet* of of **A** and **B** is the zero-one matrix with (i,j)th entry $a_{ij} \wedge b_{ij}$. The *meet* of **A** and **B** is denoted by $\mathbf{A} \wedge \mathbf{B}$.

Joins and Meets of 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

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

The meet of **A** and **B** is

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

Boolean Product of Zero-One Matrices

Definition: Let **A** = $[a_{ij}]$ be an $m \times k$ zero-one matrix and **B** = $[b_{ij}]$ be a $k \times n$ zero-one matrix. The *Boolean product* of **A** and **B**, denoted by **A** $\odot$ **B**, is the $m \times n$ zero-one matrix with (i,j)th entry

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

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

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

Continued on next slide →

Boolean Product of Zero-One Matrices

S**olution**: The Boolean product **A** $\odot$ **B** is given by

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

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

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

Boolean Powers of Zero-One Matrices

Definition: Let **A** be a square zero-one matrix and let r be a positive integer. The r^{th} Boolean power of **A** is the Boolean product of r factors of **A**, denoted by $\mathbf{A}^{[r]}$. Hence,

$$\mathbf{A}^{[r]} = \underbrace{\mathbf{A} \odot \mathbf{A} \odot \ldots \odot \mathbf{A}}_{r \text{ times}}.$$

We define $\mathbf{A}^{[0]}$ to be $\mathbf{I}_n$.

Boolean Powers of Zero-One Matrices

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:

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

$$\mathbf{A}^{[4]} = \mathbf{A}^{[3]} \odot \mathbf{A} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \qquad \mathbf{A}^{[5]} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

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

Algorithms (Ch. 3.1)

- Properties of Algorithms
- Algorithms for Searching and Sorting
- Greedy Algorithms
- Halting Problem

Problems and Algorithms

- In many domains there are key general problems that ask for output with specific properties when given valid input.
- The first step is to precisely state the problem, using the appropriate structures to specify the input and the desired output.
- We then solve the general problem by specifying the steps of a procedure that takes a valid input and produces the desired output. This procedure is called an *algorithm*.

Algorithms

Abu Ja’far Mohammed Ibin Musa Al-Khowarizmi (780-850)

Definition: An *algorithm* is a finite set of precise instructions for performing a computation or for solving a problem.

Example: Describe an algorithm for finding the maximum value in a finite sequence of integers.

Solution: Perform the following steps:

1. Set the temporary maximum equal to the first integer in the sequence.
2. Compare the next integer in the sequence to the temporary maximum.
   - If it is larger than the temporary maximum, set the temporary maximum equal to this integer.
3. Repeat the previous step if there are more integers. If not, stop.
4. When the algorithm terminates, the temporary maximum is the largest integer in the sequence.

Specifying Algorithms

- Algorithms can be specified in different ways. Their steps can be described in English or in *pseudocode.*
- Pseudocode is an intermediate step between an English language description of the steps and a coding of these steps using a programming language.
- The form of pseudocode we use is specified in Appendix 3. It uses some of the structures found in popular languages such as C++ and Java.
- Programmers can use the description of an algorithm in pseudocode to construct a program in a particular language.
- Pseudocode helps us analyze the time required to solve a problem using an algorithm, independent of the actual programming language used to implement algorithm.

Properties of Algorithms

- *Input*: An algorithm has input values from a specified set.
- *Output*: From the input values, the algorithm produces the output values from a specified set. The output values are the solution.
- *Correctness*: An algorithm should produce the correct output values for each set of input values.
- *Finiteness*: An algorithm should produce the output after a finite number of steps for any input.
- *Effectiveness*: It must be possible to perform each step of the algorithm correctly and in a finite amount of time.
- *Generality*: The algorithm should work for all problems of the desired form.

Finding the Maximum Element in a Finite Sequence

- The algorithm in pseudocode:

```
procedure max(a_1, a_2, …., a_n: integers)
   max := a_1
   for i := 2 to n
       if max < a_i then max := a_i
   return max {max is the largest element}
```

- Does this algorithm have all the properties listed on the previous slide?

Some Example Algorithm Problems

- Three classes of problems will be studied in this section.
  1. *Searching Problems*: finding the position of a particular element in a list.
  2. *Sorting Problems*: putting the elements of a list into increasing order.
  3. *Optimization Problems*: determining the optimal value (maximum or minimum) of a particular quantity over all possible inputs.

Searching Problems

Definition: The general *searching problem* is to locate an element *x* in the list of distinct elements $a_1, a_2, \ldots, a_n$, or determine that it is not in the list.

- The solution to a searching problem is the location of the term in the list that equals *x* (that is, *i* is the solution if $x = a_i$) or 0 if *x* is not in the list.
- For example, a library might want to check to see if a patron is on a list of those with overdue books before allowing him/her to checkout another book.
- We will study two different searching algorithms; linear search and binary search.

Linear Search Algorithm

- The linear search algorithm locates an item in a list by examining elements in the sequence one at a time, starting at the beginning.
  - First compare x with a_1. If they are equal, return the position 1.
  - If not, try a_2. If $x = a_2$, return the position 2.
  - Keep going, and if no match is found when the entire list is scanned, return 0.

> **procedure** *linear search*(x:integer, $a_1, a_2, \ldots, a_n$: distinct integers)
> $i := 1$
> **while** ($i \leq n$ and $x \neq a_i$)
> $i := i + 1$
> **if** $i \leq n$ **then** *location* $:= i$
> **else** *location* $:= 0$
> **return** *location*{*location* is the subscript of the term that equals x, or is 0 if x is not found}

  - Why is this referred to as a "linear" algorithm?

Binary Search

- Assume the input is a list of items in increasing order.
- The algorithm begins by comparing the element to be found with the middle element.
  - If the middle element is lower, the search proceeds with the upper half of the list.
  - If it is not lower, the search proceeds with the lower half of the list (through the middle position).
- Repeat this process until we have a list of size 1.
  - If the element we are looking for is equal to the element in the list, the position is returned.
  - Otherwise, 0 is returned to indicate that the element was not found.
- In Section 3.3, we show that the binary search algorithm is much more efficient than linear search.

Binary Search

- Here is a description of the binary search algorithm in pseudocode.

procedure binary search(x: integer, $a_1, a_2, \ldots, a_n$: increasing integers)

$i := 1$ {i is the left endpoint of interval}

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

while $i < j$

  $m := \lfloor (i + j)/2 \rfloor$

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

  else $j := m$

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

else *location* $:= 0$

return *location*{location is the subscript i of the term a_i equal to x, or 0 if x is not found}

Binary Search

Example: The steps taken by a binary search for 19 in the list:

1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22

1. The list has 16 elements, so the midpoint is 8. The value in the 8^{th} position is 10. Since $19 > 10$, further search is restricted to positions 9 through 16.

   1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22

2. The midpoint of the list (positions 9 through 16) is now the 12^{th} position with a value of 16. Since $19 > 16$, further search is restricted to the 13^{th} position and above.

   1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22

Binary Search

3. The midpoint of the current list is now the 14th position with a value of 19. Since $19 \ngtr 19$, further search is restricted to the portion from the 13th through the 14th positions .

   1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22

4. The midpoint of the current list is now the 13th position with a value of 18. Since 19> 18, search is restricted to the portion from the 18th position through the 18th.

   1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22

5. Now the list has a single element and the loop ends. Since 19=19, the location 16 is returned.

Sorting

- To *sort* the elements of a list is to put them in increasing order (numerical order, alphabetic, and so on).
- Sorting is an important problem because:
  - A nontrivial percentage of all computing resources are devoted to sorting different kinds of lists, especially applications involving large databases of information that need to be presented in a particular order (e.g., by customer, part number, etc.).
  - An amazing number of fundamentally different algorithms have been invented for sorting. Their relative advantages and disadvantages have been studied extensively.
  - Sorting algorithms are useful to illustrate the basic notions of computer science.
- A variety of sorting algorithms are studied in this book; binary, insertion, bubble, selection, merge, quick, and tournament.
- In Section 3.3, we'll study the amount of time required to sort a list using the sorting algorithms covered in this section.

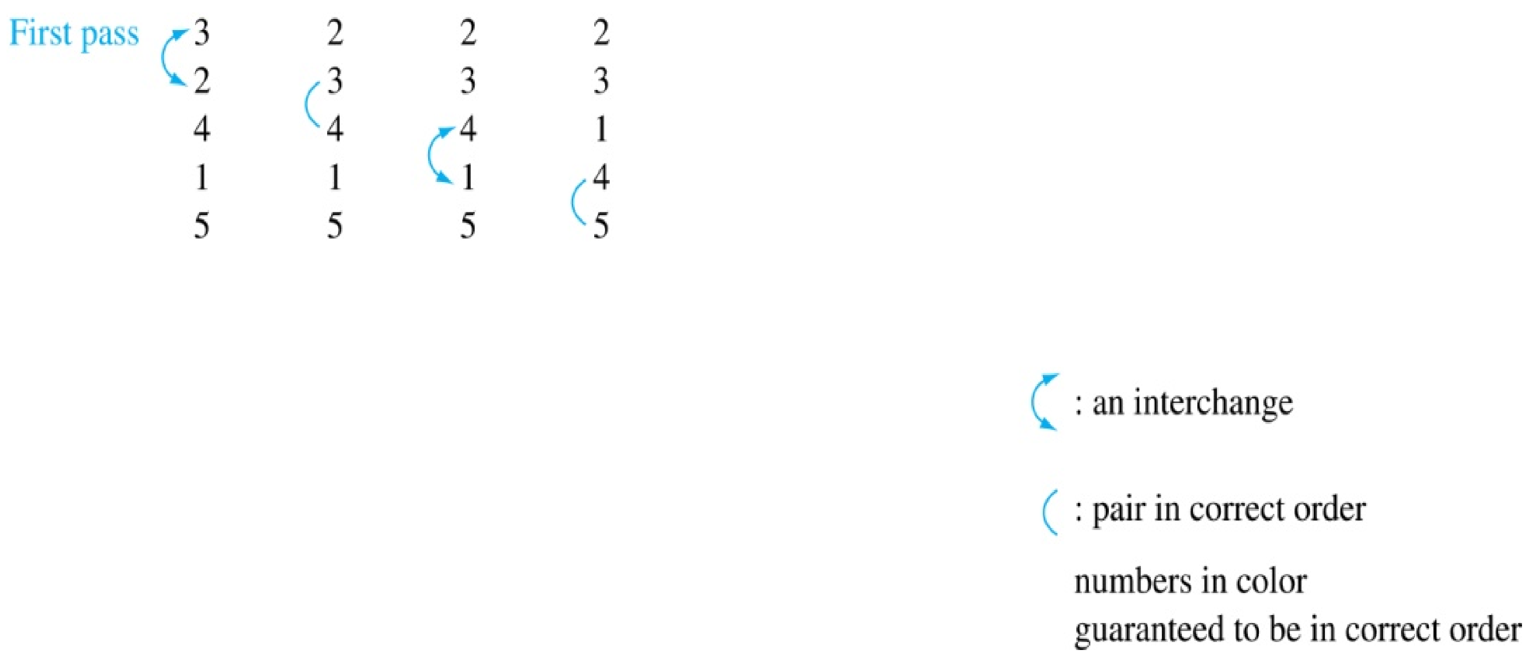
Bubble Sort

- *Bubble sort* makes multiple passes through a list. Every pair of elements that are found to be out of order are interchanged.

procedure *bubblesort*($a_1,\ldots,a_n$: real numbers
with $n \geq 2$)
  for $i := 1$ to $n - 1$
    for $j := 1$ to $n - i$
      if $a_j > a_{j+1}$ **then** interchange a_j and a_{j+1}
{$a_1,\ldots, a_n$ is now in increasing order}

Bubble Sort

Example: Show the steps of bubble sort with 3 2 4 1 5



- At the first pass the largest element has been put into the correct position
- At the end of the second pass, the 2nd largest element has been put into the correct position.
- In each subsequent pass, an additional element is put in the correct position.

Insertion Sort

- *Insertion sort* begins with the 2nd element. It compares the 2nd element with the 1st and puts it before the first if it is not larger.
- Next the 3rd element is put into the correct position among the first 3 elements.
- In each subsequent pass, the $n+1$st element is put into its correct position among the first $n+1$ elements.
- Linear search is used to find the correct position.

procedure *insertion sort* ($a_1,\ldots,a_n$: real numbers with $n \geq 2$)
  for $j := 2$ to n
    $i := 1$
    while $a_j > a_i$
      $i := i + 1$
    $m := a_j$
    for $k := 0$ to $j - i - 1$
      $a_{j-k} := a_{j-k-1}$
    $a_i := m$
{Now $a_1,\ldots,a_n$ is in increasing order}

Insertion Sort

Example: Show all the steps of insertion sort with the input: 3 2 4 1 5

i. 2 3 4 1 5 (*first two positions are interchanged*)
ii. 2 3 4 1 5 (*third element remains in its position*)
iii. 1 2 3 4 5 (*fourth is placed at beginning*)
iv. 1 2 3 4 5 (*fifth element remains in its position*)