Data Structure

تركيب بيانات
الفرقة الثالثة علوم إحصاء وعلوم الحاسب

By

Dr. Reda Elbarougy

د/ رضا الباروجى

Lecturer of computer sciences
In Mathematics Department
Faculty of Science
Damietta University

رقم المحاضرة

رقم المحاضرة	التاريخ
المحاضرة 1	2020-02
المحاضرة 2	2020-03-03
المحاضرة 3	2020-03-10
المحاضرة 4	2020-03-17
المحاضرة 5	
المحاضرة 6	
المحاضرة 7	
المحاضرة 8	
المحاضرة 9	
المحاضرة 10	
المحاضرة 11	

Arrays, Records and Pointers

Chapter 4: Arrays, Records and Pointers

1. Introduction
2. Linear Arrays and Some Examples
3. Representation of Linear Arrays in Memory
4. Traversing Linear Arrays
    1. Algorithm 4.1: traversing a linear array
    2. Algorithm 4.1’: traversing a linear array
5. Inserting and Deleting
    1. Algorithm 4.2: Inserting into a linear array
    2. Algorithm 4.3: Deleting from a linear array
6. Sorting; Bubble Sort
    1. Algorithm 4.4: Bubble sort
    2. Complexity of the bubble sort algorithm

Chapter 4: Arrays, Records and Pointers

7. Searching: Linear search
    1. Algorithm 4.5: linear search
    2. Complexity of the linear search algorithm
8. Searching: Binary Search
    1. Algorithm 4.6: Binary search
    2. Complexity of the binary search algorithm
    3. Limitation of the Binary Search Algorithm

Categories of Data Structure

Data structure can be classified in to major types:

- Linear Data Structure (Chapter 4, 5 and 6)
- Non-linear Data Structure (Chapter 7)

Linear Data Structure:

A data structure is said to be **linear** if its elements form a sequence, or, in other words, a linear list.

There are basically two ways of representing such linear structure in memory.

a) ***One way is to have the linear relationships between the elements represented*** *by means of sequential memory location. These linear structures are called* **arrays.**

b) ***The other way is to have the linear relationship between the elements*** *represented by means of pointers or links.* These linear structures are called **linked lists.**

Common examples of linear data structure

The common examples of linear data structure are

- **Arrays**
- **Queues**
- **Stacks**
- **Linked lists**

Non-linear Data Structure

This structure is mainly used to represent data containing a hierarchical relationship between elements.

–e.g. **graphs**, family **trees** and table of contents.

Operations on linear structure

Operation on linear structure (array or linked list):

a) Traversing: Processing each element in the list.

b) Searching: Finding the location of a particular element in with a given value or the record with a given key.

c) Insertion: Adding a new element to the list.

d) Deletion: Removing an element from a list.

e) Sorting: Arranging the elements in some type of order (Ascending / Descending).

f) Merging: Combining two lists into a single list.

Linear Arrays

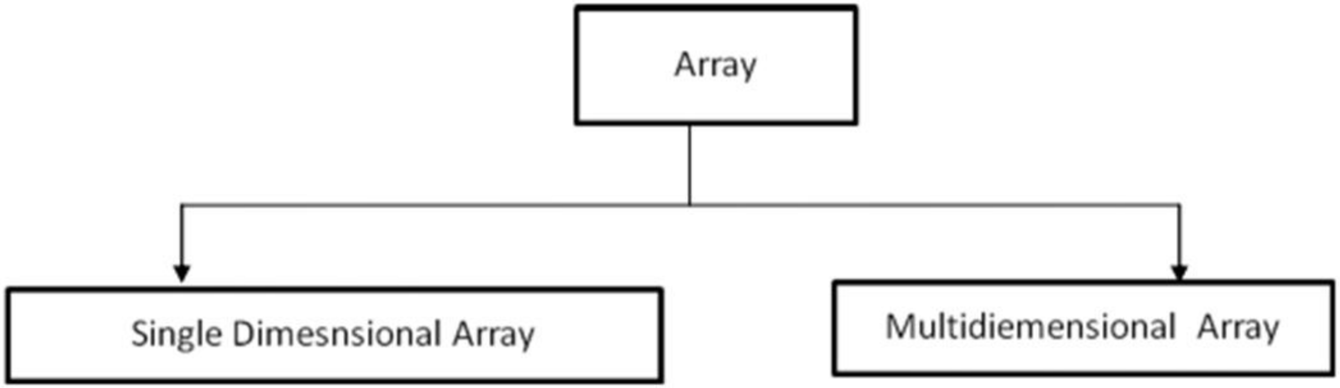
- The simplest type of data structure is a linear (or one dimensional) array.
- A linear array is a list of a finite number n of similar data elements such that:

a) The elements of the array are referenced respectively by an index set consisting of n consecutive numbers, usually 1, 2, 3 . . . . n.

b) The elements of the array are stored respectively in a successive memory locations.

- if we choose the name A for the array, then the elements of A are denoted by subscript notation

Arrays



Array

Single Dimesnsional Array

Multidiemensional Array

Ex:- STUDENT

Ram
Amit
Rohan
Rita
Sunil

MARKS

Sub Rollno	Sub 1	Sub 2	Sub 3	Sub 4
1	50	60	70	55
2	40	20	90	75
⋮				
⋮				
50				

Linear Arrays

Advantage :-

- Structure is simple .
- Arrays are easy to traverse ,search & sort.

Disadvantages:-

- Insertion & deletion is difficult .It involves data movement.

Linear Arrays

If we choose the name A for the array, then the elements of A are denoted by subscript notation.

The number k in A[k] is called subscript or index

A_1, A_2, A_3 . . . . A_n
or by the parenthesis notation
A (1), A (2), A (3) . . . . A (n)
or by the bracket notation
A [1], A [2], A [3] . . . . A [n]

Example:

A linear array **A[8]** consisting of numbers is pictured in following figure.

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
1	2	3	4	5	6	7	8

```
int A[8] = {1, 2, 3, 4, 5, 6, 7, 8};
```

Linear Arrays

N is called the length

Length=UB-LB+1

UB is largest index, called upper bound

LB is smallest index, called lower bound.

Note that Length=UB when LB=1

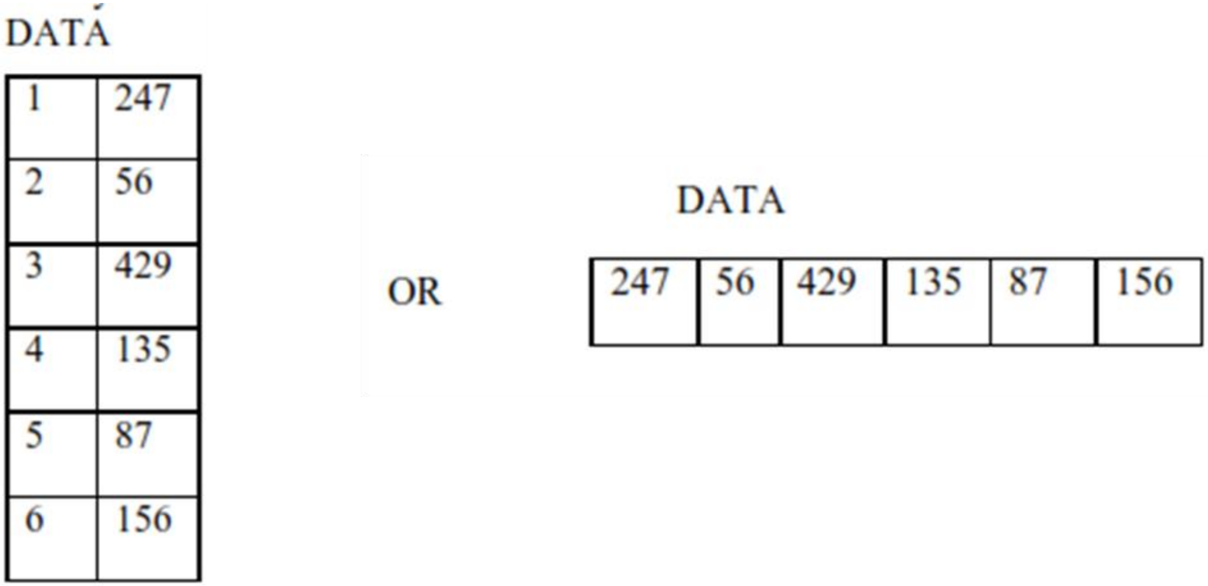
Linear Arrays: Example 4.1.

Example 4.1 (a): Let data is a six element linear array of integer such that:

DATA [1] = 247 DATA [2] = 56 DATA [3] =429

DATA [4] =135 DATA [5] = 87 DATA [6] =156

- DATA 247, 56, 429, 135, 87
- This type of array data can be pictured in the form:



DATA

1	247
2	56
3	429
4	135
5	87
6	156

OR

DATA

247	56	429	135	87	156

Linear Arrays: Example 4.1.

Example 4.1 (b): AUTO to record the number of automobiles sold each year from 1932 to 1984

AUTO[k]=number of automobiles sold in the year k

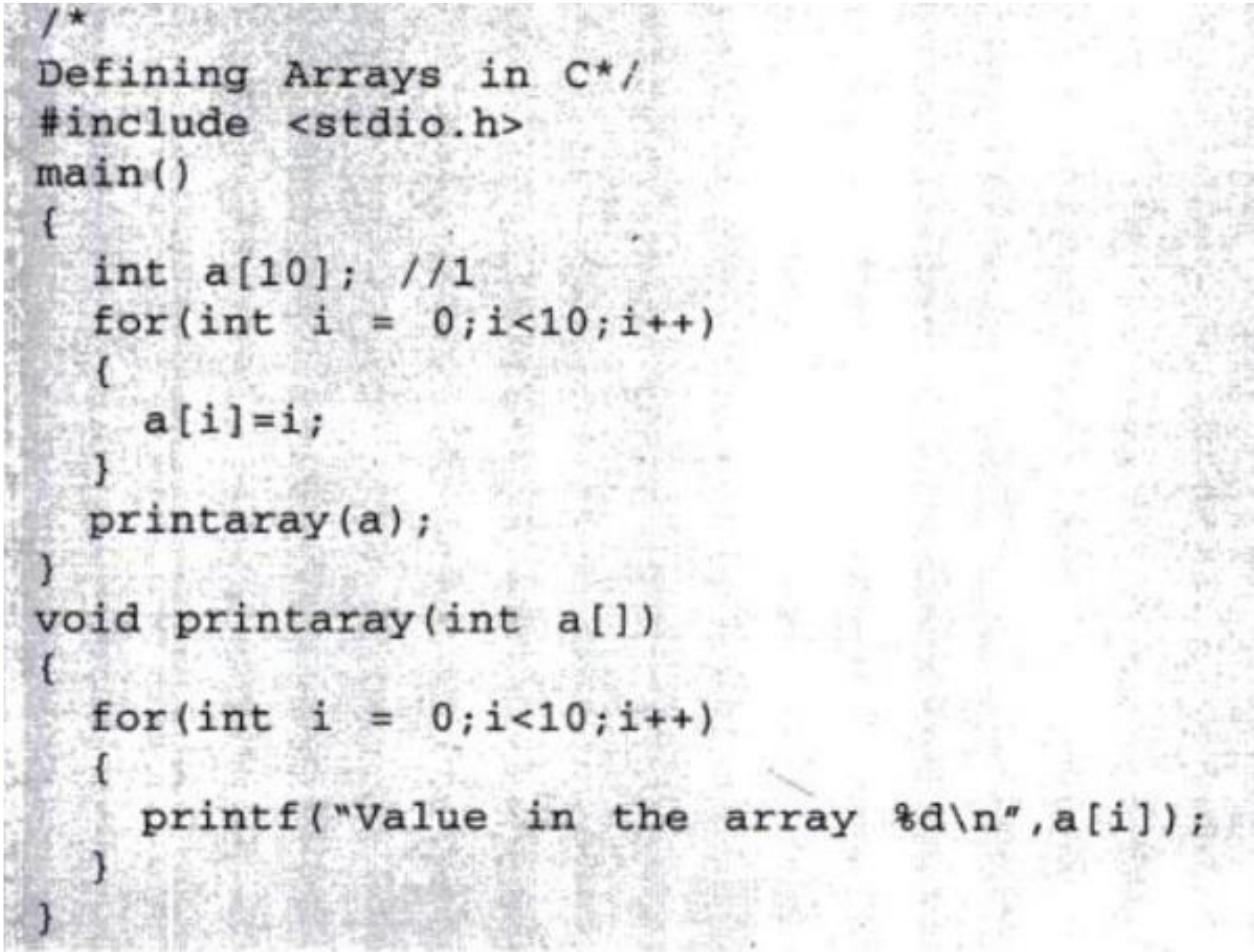
LB=1932

UB=1984

Length=UB-LB+1=1984-1930+1=55

Index are integers from 1932 to 1984

Linear Arrays: Example 4.1.



```
/*
Defining Arrays in C*/
#include <stdio.h>
main()
{
  int a[10]; //1
  for(int i = 0;i<10;i++)
  {
    a[i]=i;
  }
  printaray(a);
}
void printaray(int a[])
{
  for(int i = 0;i<10;i++)
  {
    printf(“Value in the array %d\n”,a[i]);
  }
}
```

Linear Arrays

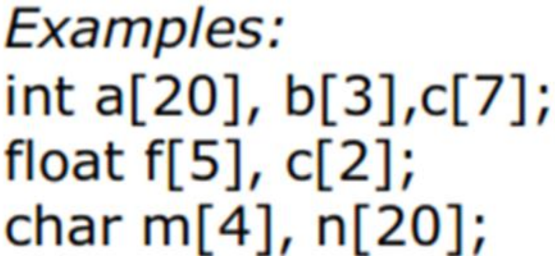
Each programing language has its own rules for dealing arrays, each such declaration must give, implicitly or explicitly, three items of information's:

1. The name of the array
2. The data type of the array and
3. The index set of the array

- **Declaration of the Arrays:** Any array declaration contains:
  1. the array name,
  2. the element type and
  3. the array size.

Linear Arrays

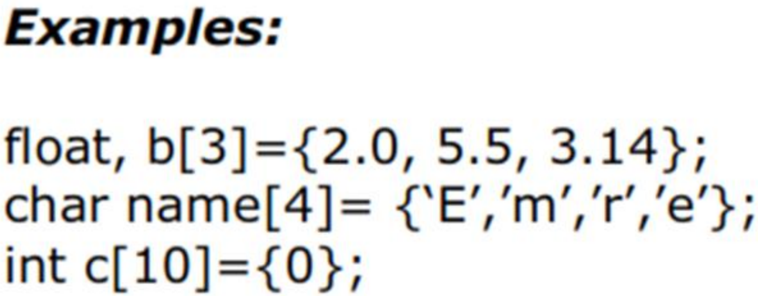
- Declaration of the Arrays: Any array declaration contains: the array name, the element type and the array size.



Examples:

```
int a[20], b[3],c[7];
float f[5], c[2];
char m[4], n[20];
```

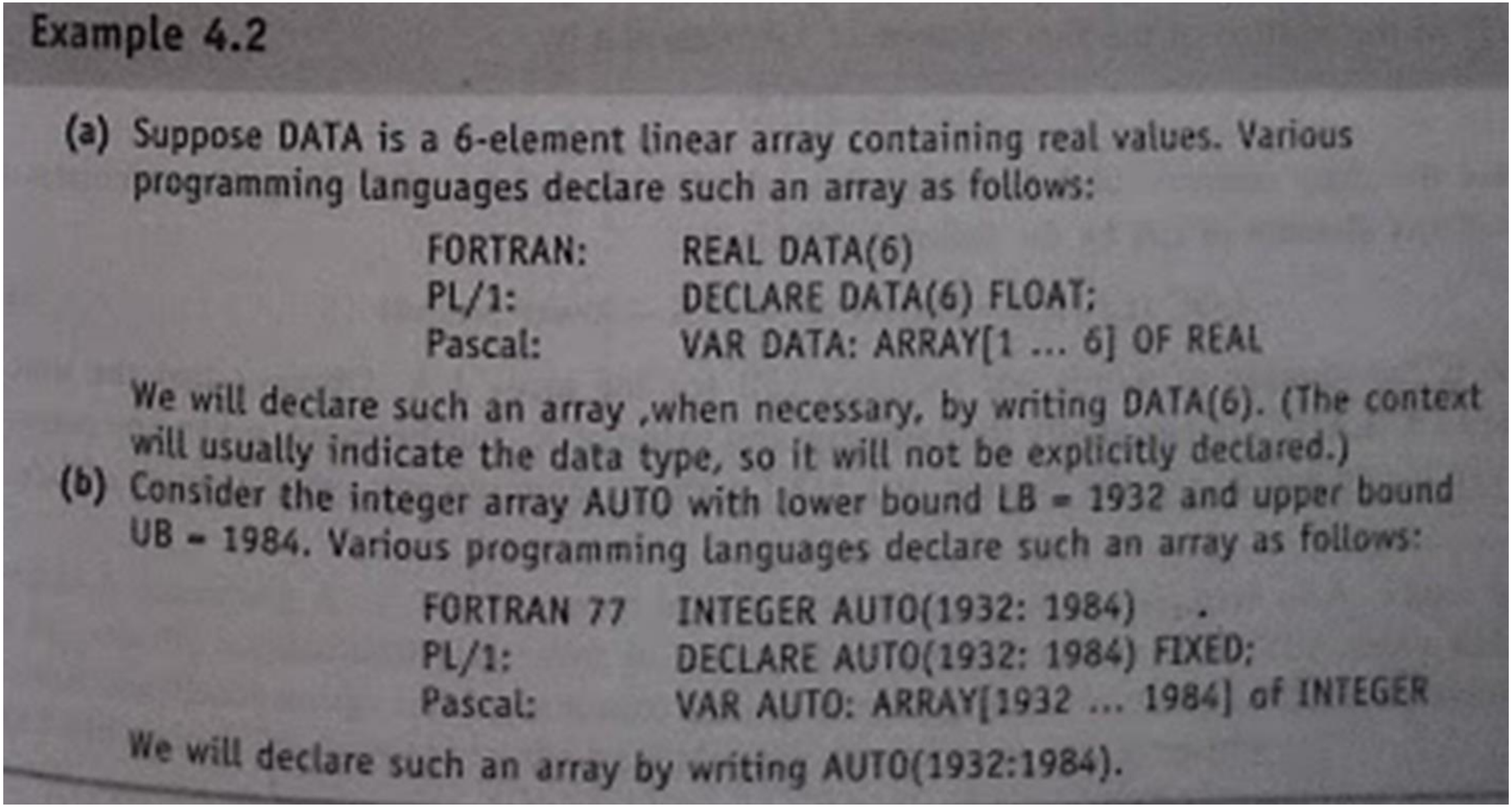
- Initialization of an array is the process of assigning initial values. Typically declaration and initialization are combined.



Examples:

```
float, b[3]={2.0, 5.5, 3.14};
char name[4]= {‘E’,’m’,’r’,’e’};
int c[10]={0};
```

Example 4.2.



Example 4.2

(a) Suppose DATA is a 6-element linear array containing real values. Various programming languages declare such an array as follows:

```
FORTRAN:     REAL DATA(6)
PL/1:        DECLARE DATA(6) FLOAT;
Pascal:      VAR DATA: ARRAY[1 ... 6] OF REAL
```

We will declare such an array ,when necessary, by writing DATA(6). (The context will usually indicate the data type, so it will not be explicitly declared.)

(b) Consider the integer array AUTO with lower bound LB = 1932 and upper bound UB = 1984. Various programming languages declare such an array as follows:

```
FORTRAN 77   INTEGER AUTO(1932: 1984)
PL/1:        DECLARE AUTO(1932: 1984) FIXED;
Pascal:      VAR AUTO: ARRAY[1932 ... 1984] of INTEGER
```

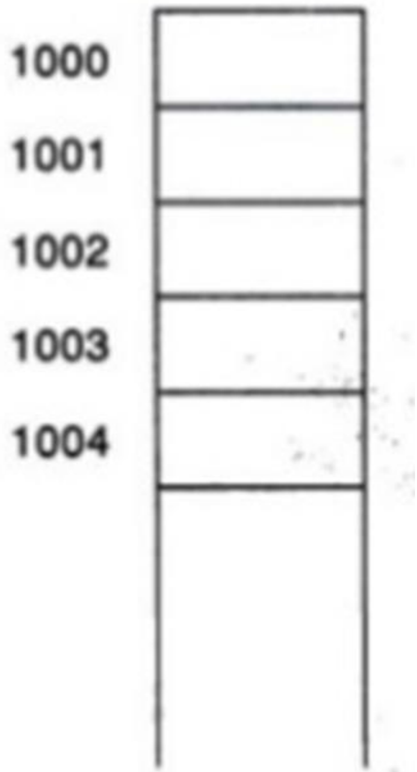
We will declare such an array by writing AUTO(1932:1984).

Allocate Memory Space

- Fortran and Pascal, allocate memory space for arrays statically the size is fixed during the program execution.
- Some programing language allow one to read an integer n and then declare an array with n elements, such programing are said to allocate memory dynamically

3. Representation of Linear Arrays in Memory

Let LA be a linear array in the memory of a computer. Recall that the memory of computer is simply a sequence of address location as in figure below;



3. Representation of Linear Arrays in Memory

LOC(LA[K])=address of the element LA[K] of the array LA

Computer only keep the addresses of the first element Base(LA) of the array.

Base(LA) is called the base address

The address of any element is calculated by

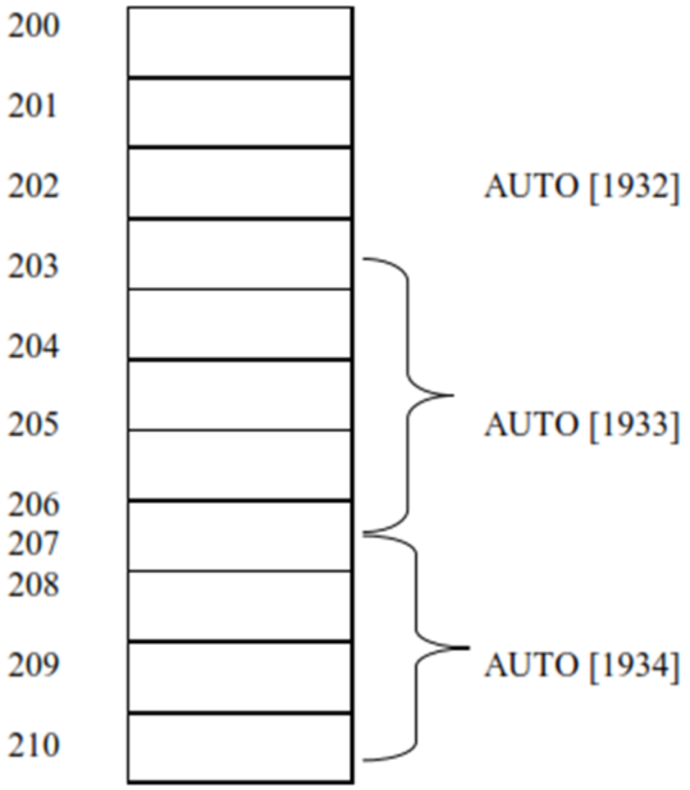
$$LOC(LA[K])=Base(LA)+W(K-LB)$$

Where W is number of words per memory cell

Example 3:

Consider the array also AUTO in example 2 which record the number of automobile sold each year from 1932 through 1984. Suppose AUTO appear in memory as picture in fig. (2) i.e base AUTO = 200 and w=4 word per memory cell for AUTO.

Find the address of the element which store the info about sale in year 1965?



Example 3:

- LOC (AUTO [1932]) = 200
- LOC (AUTO [1933]) = 204
- LOC (AUTO [1934]) = 208
- The address of the array element for the year K = 1965 can be obtained by using the equation of the formula.

LOC (LA [K]) = Base (LA) + w(K – Lower bound)

LOC (LA [1965]) = 200+4(1965 – 1932)

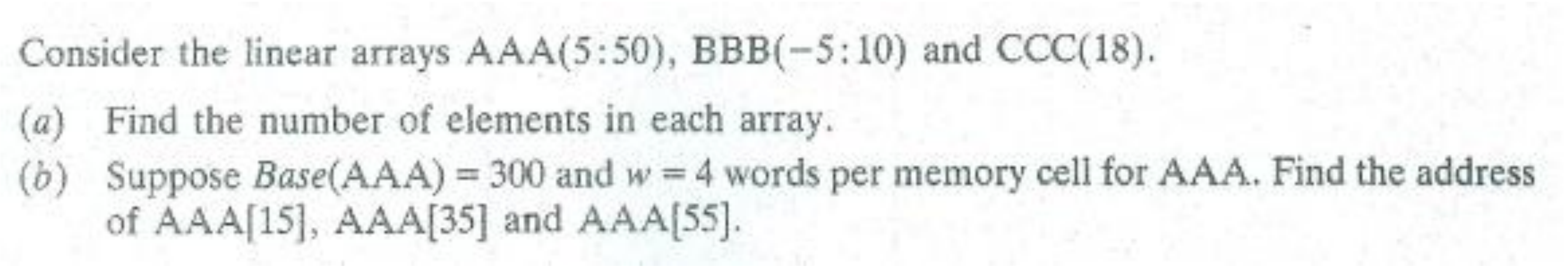
=200+4(33) = 200+132 = 332

BASE (LA) = BASE (AUTO) = 200

where w=4, K=1965, LB= 1932

➔LOC (LA [1965]) = 332.

Example 3:

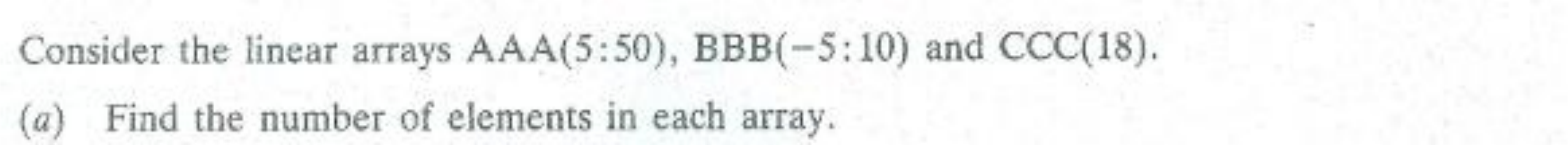


Consider the linear arrays AAA(5:50), BBB(−5:10) and CCC(18).

(*a*) Find the number of elements in each array.

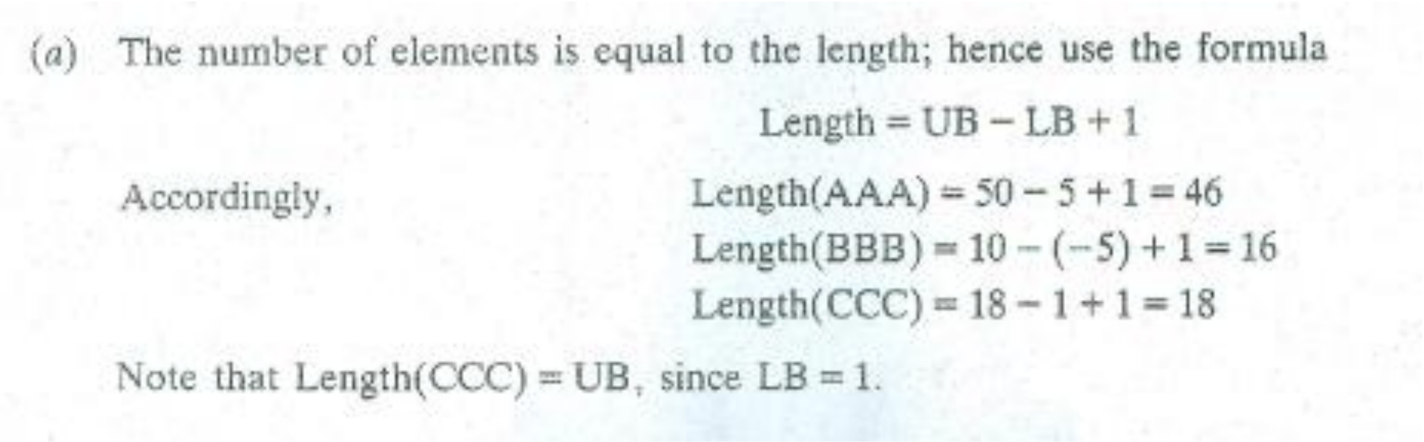
(*b*) Suppose *Base*(AAA) = 300 and $w = 4$ words per memory cell for AAA. Find the address of AAA[15], AAA[35] and AAA[55].

Example 3:



Consider the linear arrays AAA(5:50), BBB(−5:10) and CCC(18).

(*a*) Find the number of elements in each array.



(*a*) The number of elements is equal to the length; hence use the formula

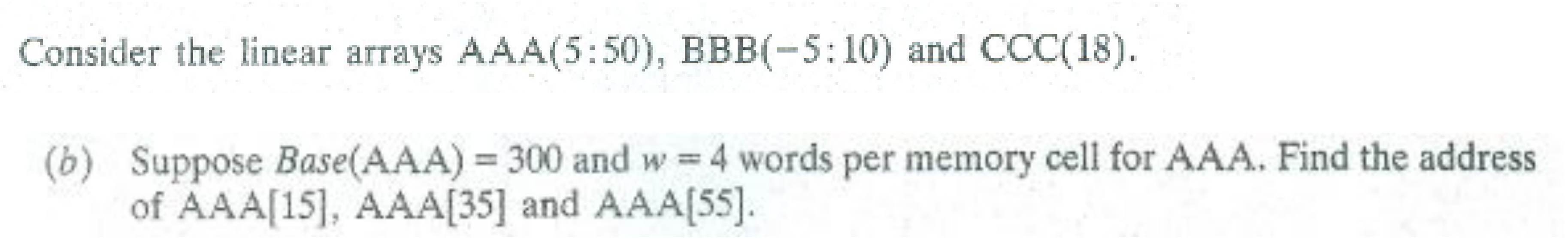
$$\text{Length} = \text{UB} - \text{LB} + 1$$

Accordingly,

$$\text{Length(AAA)} = 50 - 5 + 1 = 46$$
$$\text{Length(BBB)} = 10 - (-5) + 1 = 16$$
$$\text{Length(CCC)} = 18 - 1 + 1 = 18$$

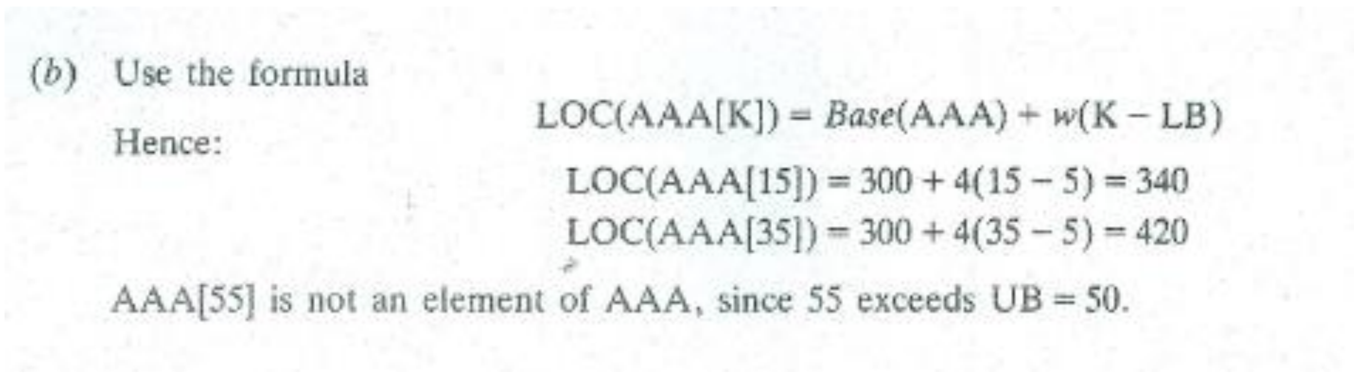
Note that Length(CCC) = UB, since LB = 1.

Example 3:



Consider the linear arrays AAA(5:50), BBB(−5:10) and CCC(18).

(*b*) Suppose *Base*(AAA) = 300 and $w = 4$ words per memory cell for AAA. Find the address of AAA[15], AAA[35] and AAA[55].



(*b*) Use the formula

$$\text{LOC}(\text{AAA}[\text{K}]) = Base(\text{AAA}) + w(\text{K} - \text{LB})$$

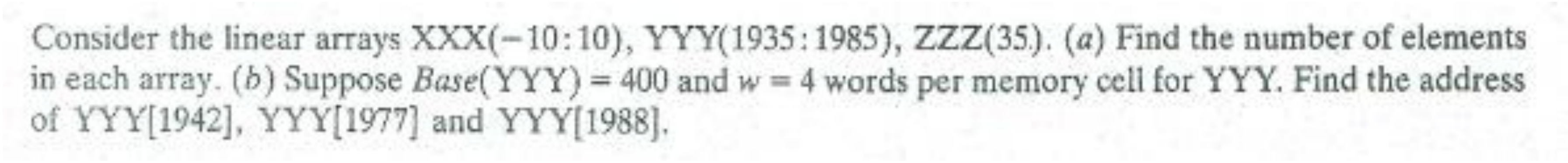
Hence:

$$\text{LOC}(\text{AAA}[15]) = 300 + 4(15 - 5) = 340$$

$$\text{LOC}(\text{AAA}[35]) = 300 + 4(35 - 5) = 420$$

AAA[55] is not an element of AAA, since 55 exceeds UB = 50.

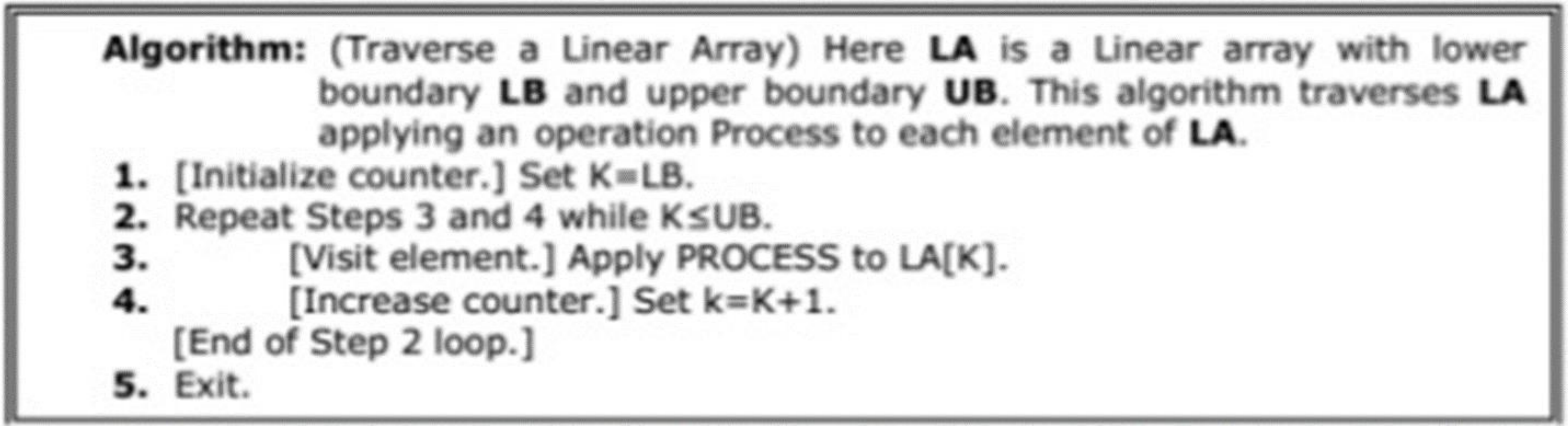
Problem



Consider the linear arrays XXX(−10:10), YYY(1935:1985), ZZZ(35). (*a*) Find the number of elements in each array. (*b*) Suppose *Base*(YYY) = 400 and $w = 4$ words per memory cell for YYY. Find the address of YYY[1942], YYY[1977] and YYY[1988].

Traversing Linear Array

Algorithm 4.1: Traversing Linear Array

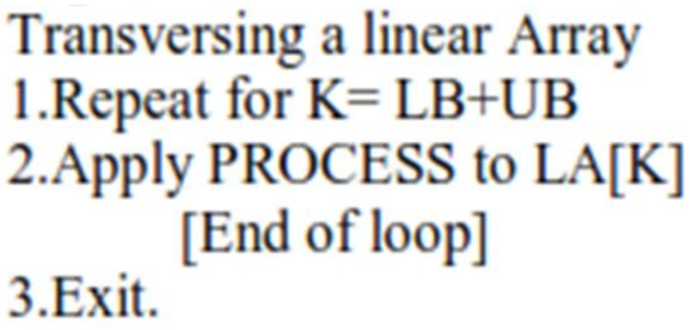


Algorithm: (Traverse a Linear Array) Here **LA** is a Linear array with lower boundary **LB** and upper boundary **UB**. This algorithm traverses **LA** applying an operation Process to each element of **LA**.

```
1.  [Initialize counter.] Set K=LB.
2.  Repeat Steps 3 and 4 while K≤UB.
3.          [Visit element.] Apply PROCESS to LA[K].
4.          [Increase counter.] Set k=K+1.
    [End of Step 2 loop.]
5.  Exit.
```

Algorithm 4.1’: Traversing Linear Array

Here LA = linear array with lower bound (LB) with upper bound (UB). This algorithm transverse LA applying an operation PROCESS to each element of LA.



```
Transversing a linear Array
1.Repeat for K= LB+UB
2.Apply PROCESS to LA[K]
        [End of loop]
3.Exit.
```

Example 4.4

Consider example 4.1(b),

(a) find the number NUM of year during which more than 300 automobile were sold.

Solution: using the algorithm

```
1) Set NUM := 0 [initialize counter]
2) Repeat for K = 1932 to 1984
    If Auto [K] >300; then set NUM: = NUM+1
End of loop
3) Loop.
```

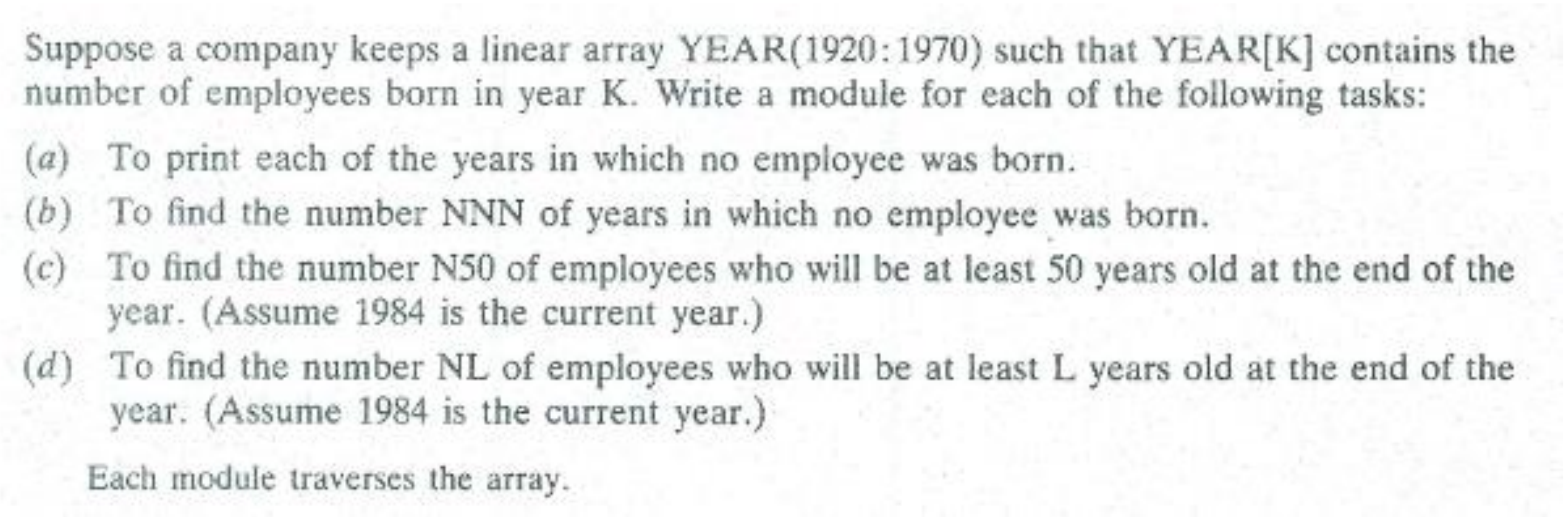
Example 4.4

(b) Print each year and the number of automobiles sold in that year.

Solution: using the algorithm

```
1) Repeat for K = 1932 to 1984:
    Write:  K, Auto [K] .
 End of loop
2) Loop.
```

Example



Suppose a company keeps a linear array YEAR(1920:1970) such that YEAR[K] contains the number of employees born in year K. Write a module for each of the following tasks:

(*a*) To print each of the years in which no employee was born.

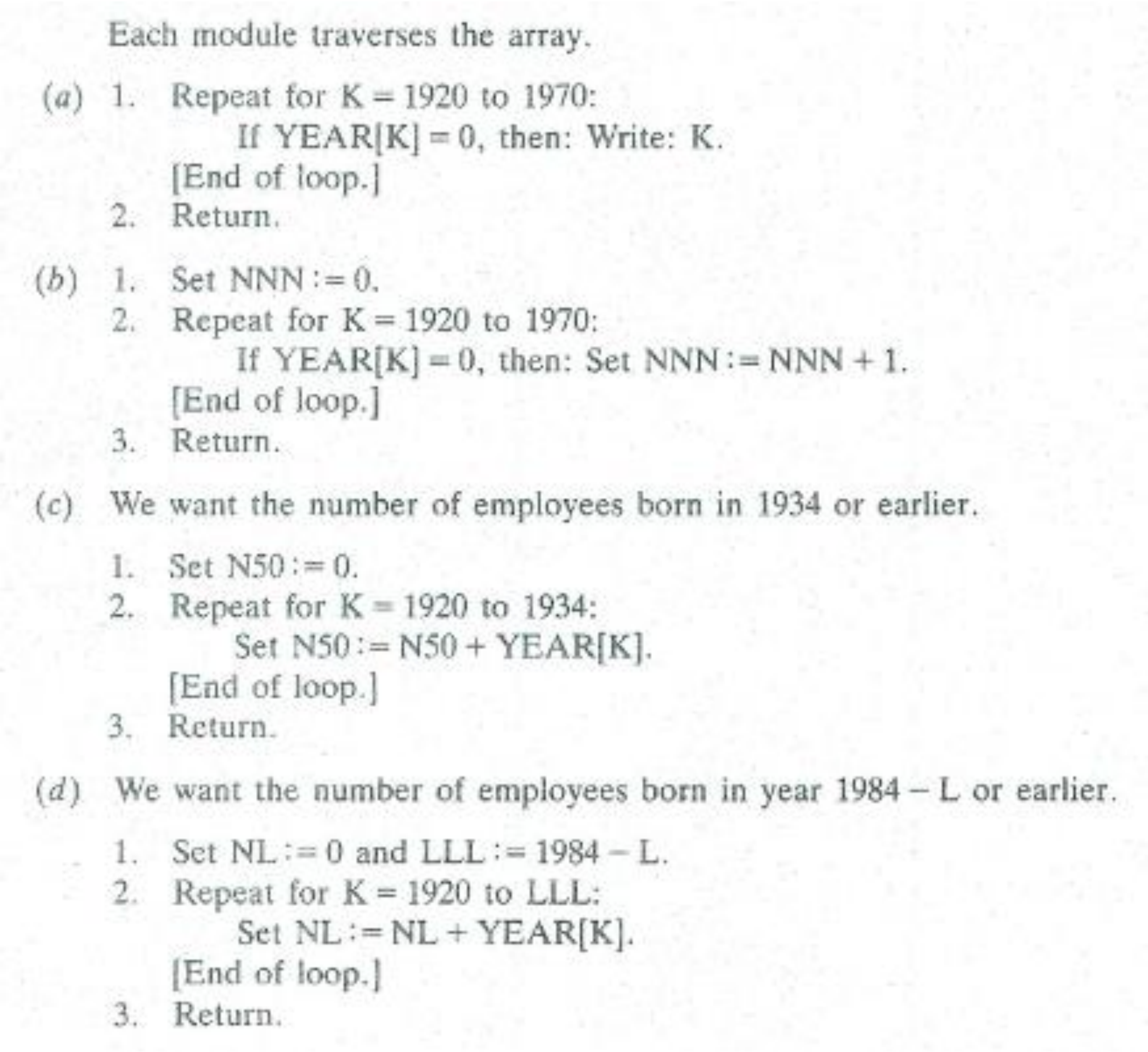
(*b*) To find the number NNN of years in which no employee was born.

(*c*) To find the number N50 of employees who will be at least 50 years old at the end of the year. (Assume 1984 is the current year.)

(*d*) To find the number NL of employees who will be at least L years old at the end of the year. (Assume 1984 is the current year.)

Each module traverses the array.

Example



Each module traverses the array.

(*a*)
```
1. Repeat for K = 1920 to 1970:
       If YEAR[K] = 0, then: Write: K.
   [End of loop.]
2. Return.
```

(*b*)
```
1. Set NNN := 0.
2. Repeat for K = 1920 to 1970:
       If YEAR[K] = 0, then: Set NNN := NNN + 1.
   [End of loop.]
3. Return.
```

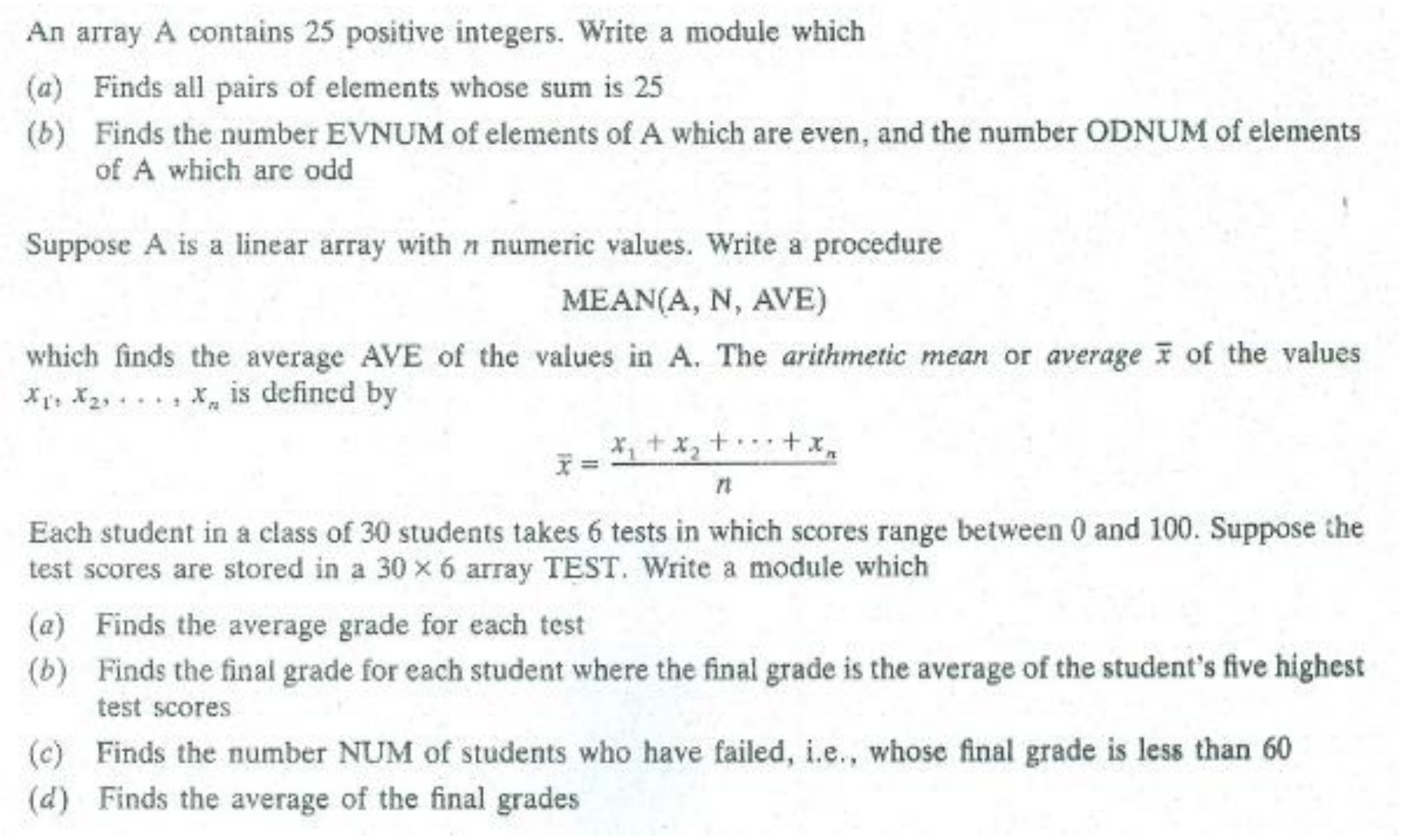
(*c*) We want the number of employees born in 1934 or earlier.

```
1. Set N50 := 0.
2. Repeat for K = 1920 to 1934:
       Set N50 := N50 + YEAR[K].
   [End of loop.]
3. Return.
```

(*d*) We want the number of employees born in year 1984 – L or earlier.

```
1. Set NL := 0 and LLL := 1984 - L.
2. Repeat for K = 1920 to LLL:
       Set NL := NL + YEAR[K].
   [End of loop.]
3. Return.
```

Problem



An array A contains 25 positive integers. Write a module which

(*a*) Finds all pairs of elements whose sum is 25

(*b*) Finds the number EVNUM of elements of A which are even, and the number ODNUM of elements of A which are odd

Suppose A is a linear array with n numeric values. Write a procedure

MEAN(A, N, AVE)

which finds the average AVE of the values in A. The *arithmetic mean* or *average* $\bar{x}$ of the values $x_1, x_2, \ldots, x_n$ is defined by

$$\bar{x} = \frac{x_1 + x_2 + \cdots + x_n}{n}$$

Each student in a class of 30 students takes 6 tests in which scores range between 0 and 100. Suppose the test scores are stored in a 30 × 6 array TEST. Write a module which

(*a*) Finds the average grade for each test

(*b*) Finds the final grade for each student where the final grade is the average of the student's five highest test scores

(*c*) Finds the number NUM of students who have failed, i.e., whose final grade is less than 60

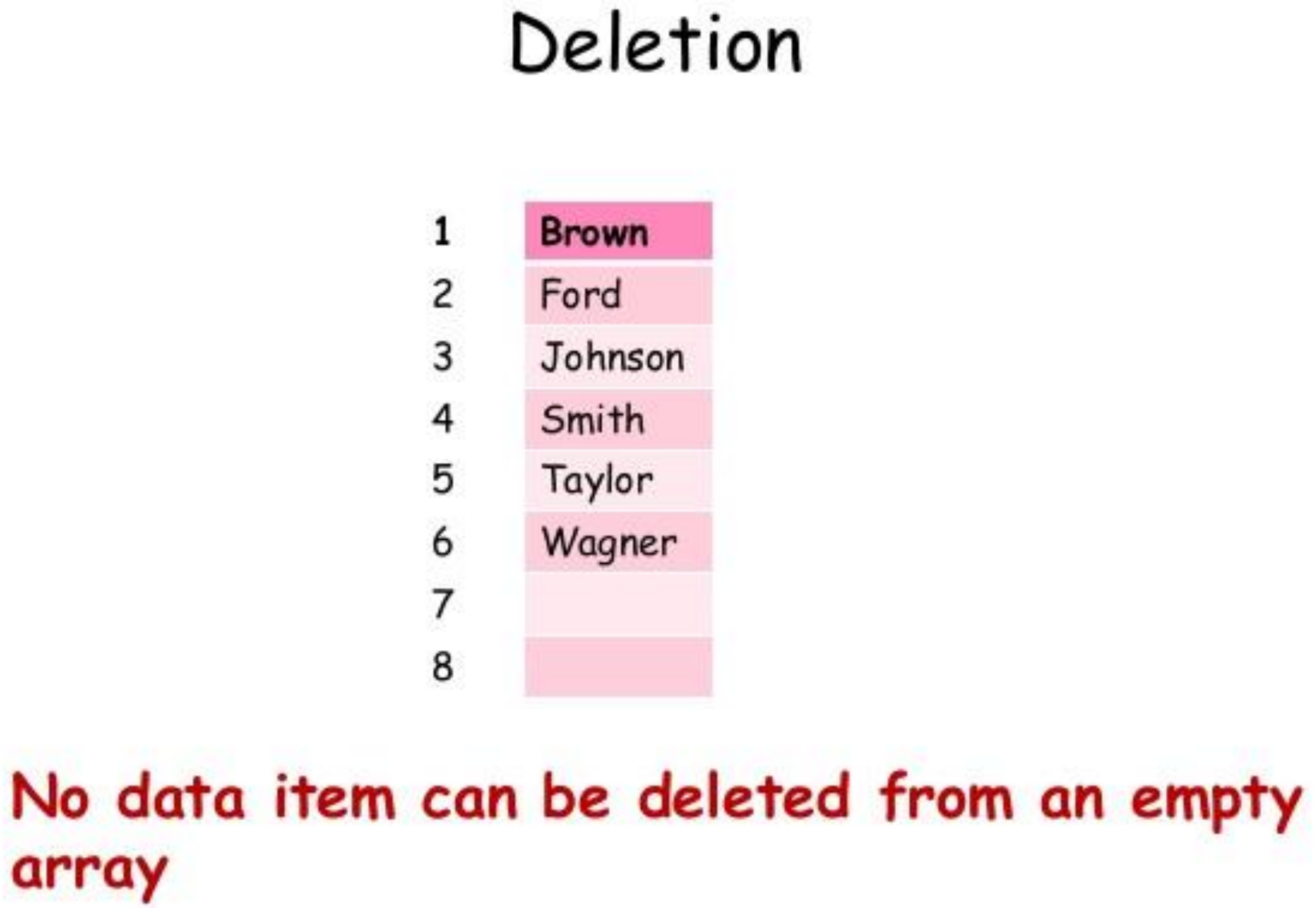
(*d*) Finds the average of the final grades

Inserting and Deleting

Inserting and Deleting

- Insert at the end can easily done
- Inserting in the middle → half of the elements move downward → increasing subscript
- Deleting from the end of a Linear Array can easily done
- Delating from the middle half of the data must move upward → decreasing the subscript

Inserting and Deleting

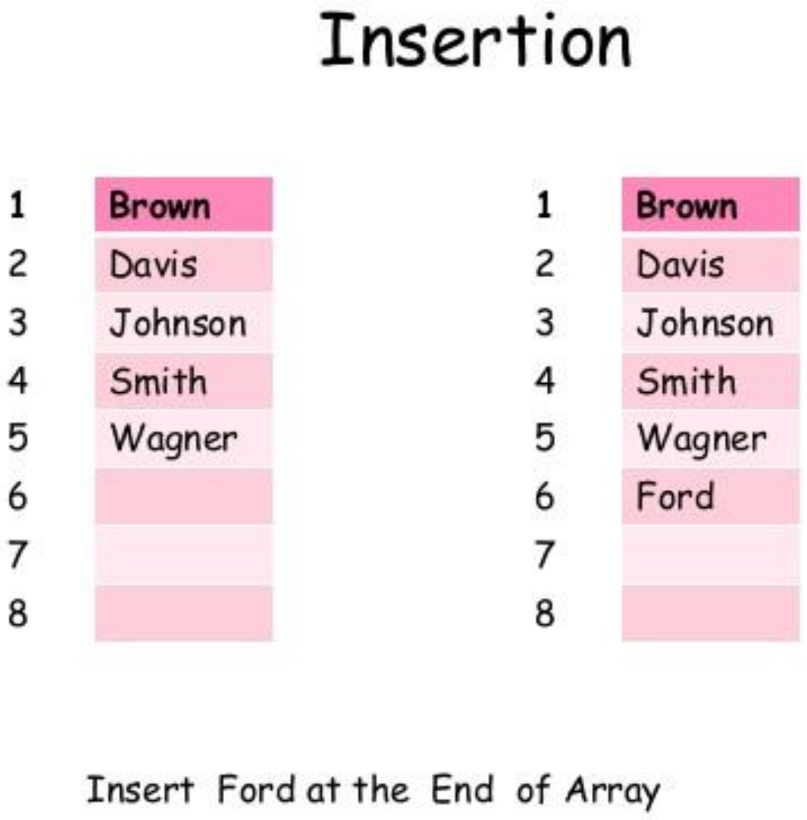


Deletion

1	**Brown**
2	Ford
3	Johnson
4	Smith
5	Taylor
6	Wagner
7	
8	

No data item can be deleted from an empty array

Inserting and Deleting



Insertion

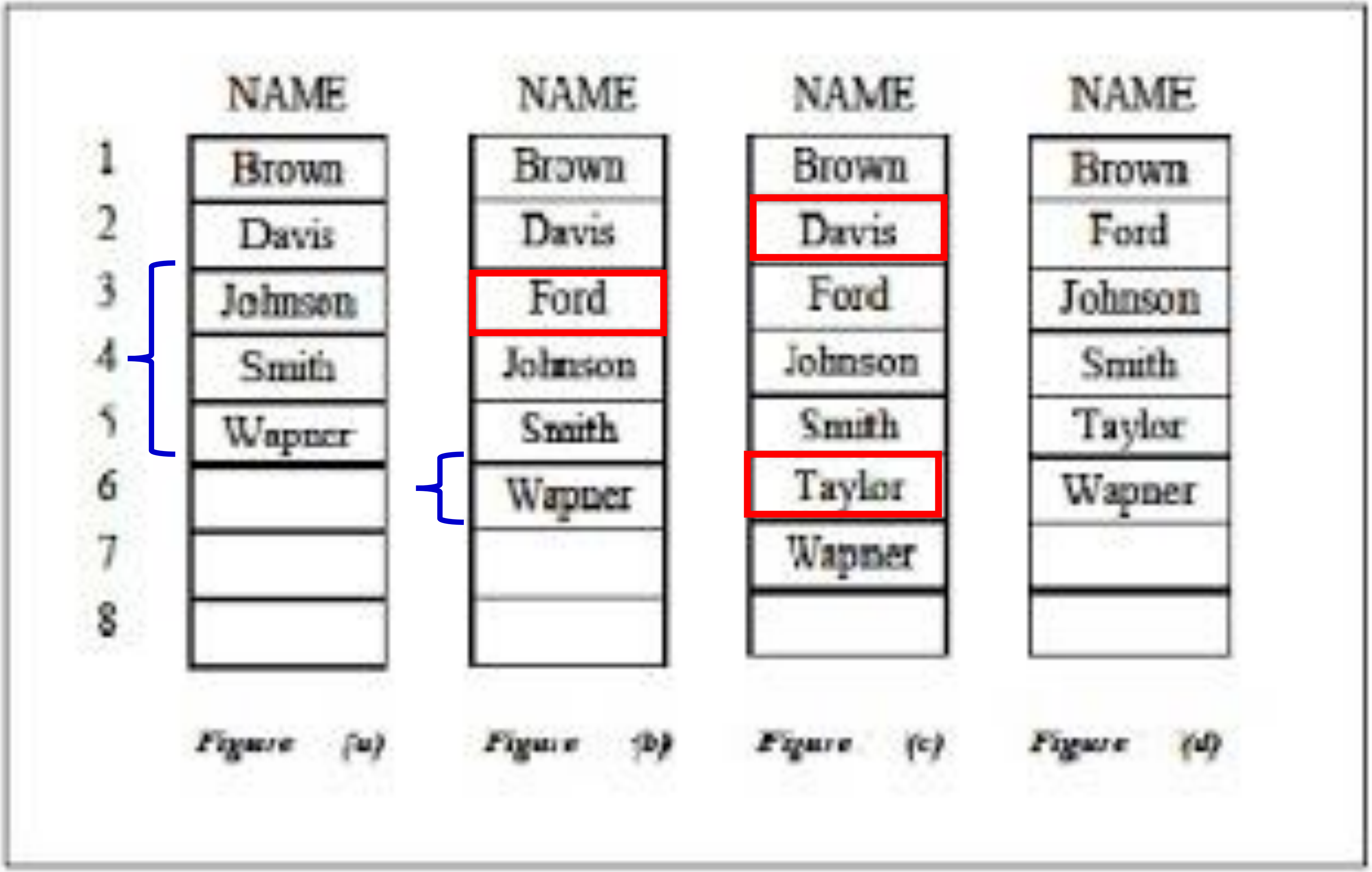
1	Brown
2	Davis
3	Johnson
4	Smith
5	Wagner
6	
7	
8	

1	Brown
2	Davis
3	Johnson
4	Smith
5	Wagner
6	Ford
7	
8	

Insert Ford at the End of Array

Inserting and Deleting

Add Ford then Add Taylor then Remove Davis



	NAME	NAME	NAME	NAME
1	Brown	Brown	Brown	Brown
2	Davis	Davis	Davis	Ford
3	Johnson	Ford	Ford	Johnson
4	Smith	Johnson	Johnson	Smith
5	Wagner	Smith	Smith	Taylor
6		Wagner	Taylor	Wagner
7			Wagner	
8				

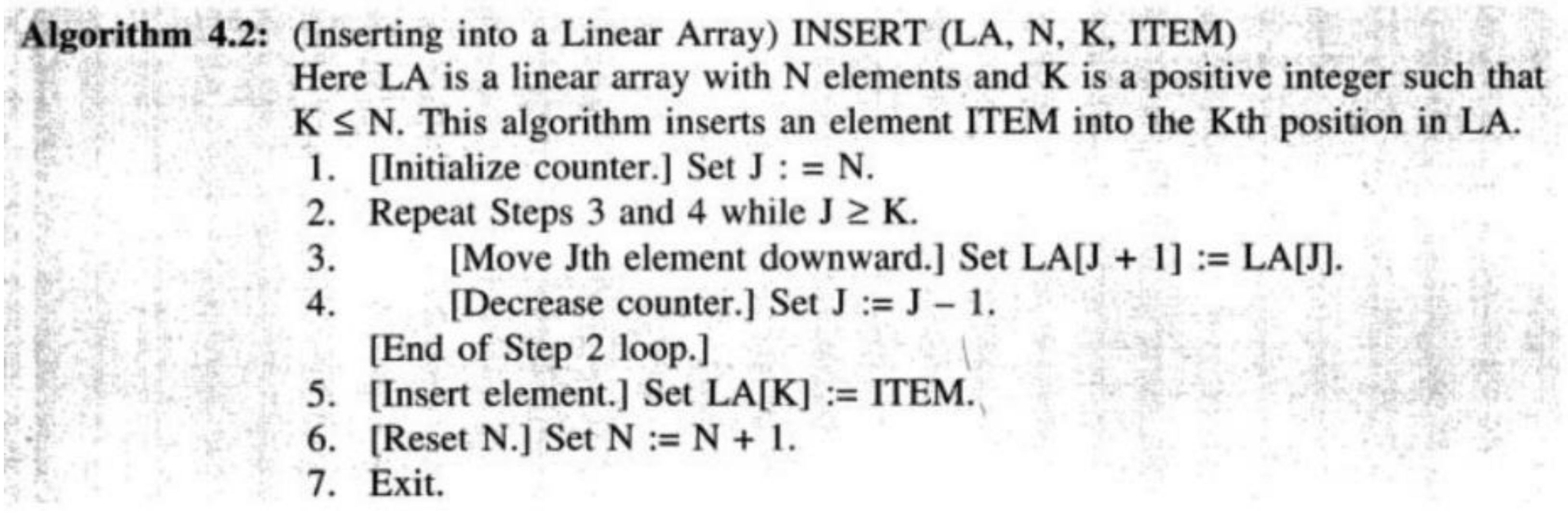
Figure (a) Figure (b) Figure (c) Figure (d)

Inserting into a Linear Array

(Algorithm:(Inserting into a linear Array)

INSERT (LA, N, K, ITEM).

Here LA is a linear array with N elements and K is a positive integer such that K ≤N. this algorithm inserts an element ITEM into the Kth position in LA.



Algorithm 4.2: (Inserting into a Linear Array) INSERT (LA, N, K, ITEM)
Here LA is a linear array with N elements and K is a positive integer such that $K \leq N$. This algorithm inserts an element ITEM into the Kth position in LA.

```
1. [Initialize counter.] Set J : = N.
2. Repeat Steps 3 and 4 while J ≥ K.
3.         [Move Jth element downward.] Set LA[J + 1] := LA[J].
4.         [Decrease counter.] Set J := J – 1.
   [End of Step 2 loop.]
5. [Insert element.] Set LA[K] := ITEM.
6. [Reset N.] Set N := N + 1.
7. Exit.
```

Deleting from Linear Array

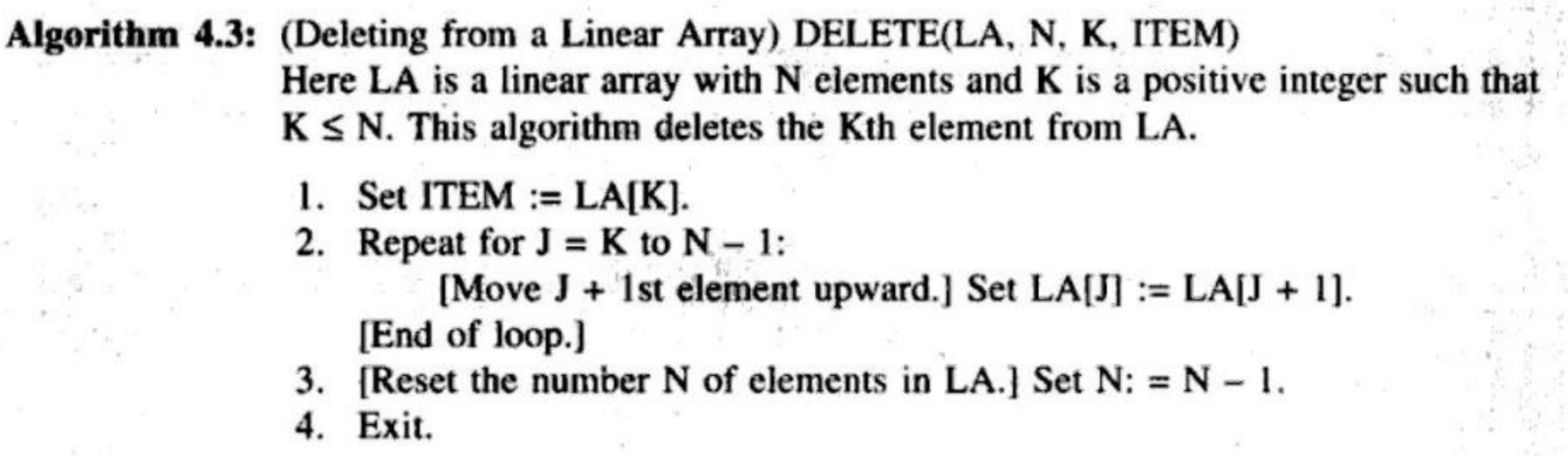
(Deleting from a Linear Array) DELETE (LA, N, K, ITEM)

Here LA is a Linear Array with N element and K is positive integer such that K≤ N.

This algorithm deletes the kth element from LA

```
1.Set ITEM := LA[K]
2.Repeat for J = K to N-1
   [Move $J^{th}$ element  upward. ] Set LA [J]:= LA [J+1]
   [End of loop]
3. [Reset the number N of elements in LA] Set N:= N-1
4.Exit.
```

Deleting from Linear Array



Algorithm 4.3: (Deleting from a Linear Array) DELETE(LA, N, K, ITEM)
Here LA is a linear array with N elements and K is a positive integer such that $K \leq N$. This algorithm deletes the Kth element from LA.

```
1. Set ITEM := LA[K].
2. Repeat for J = K to N – 1:
          [Move J + 1st element upward.] Set LA[J] := LA[J + 1].
   [End of loop.]
3. [Reset the number N of elements in LA.] Set N: = N – 1.
4. Exit.
```

Sorting in Linear Array:

Sorting an array is the ordering the array elements in ascending (increasing from min to max)

Or descending (decreasing – from max to min) order.

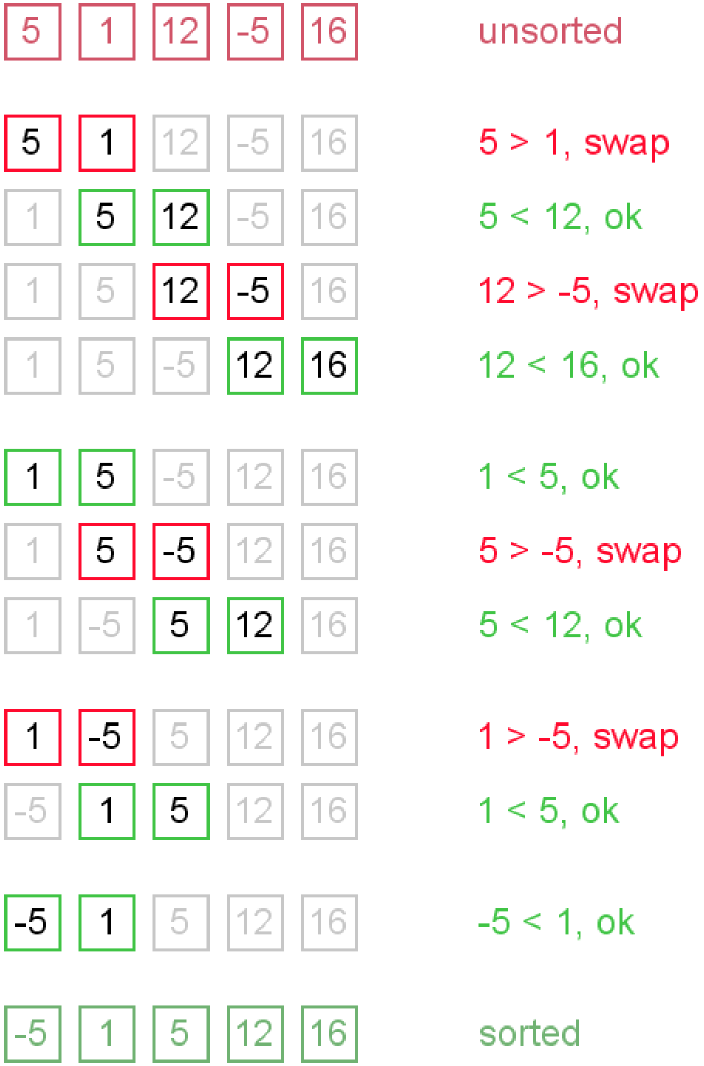
- Example:
- {2 1 5 7 4 3} → {1, 2, 3, 4, 5,7} ascending order
- {2 1 5 7 4 3} → {7,5, 4, 3, 2, 1} descending order

Bubble sort

- The technique we use is called “Bubble Sort”→
- The bigger value gradually bubbles their way up to the top of array like air bubble rising in water,
- While the small values sink to the bottom of array.
- This technique is to make several passes through the array.
- On each pass, successive pairs of elements are compared.
- If a pair is in increasing order (or the values are identical), we leave the values as they are. If a pair is in decreasing order, their values are swapped in the array.

Example 4.2.

Example. **Sort {5, 1, 12, -5, 16} using bubble sort.**



5 1 12 -5 16 unsorted

5 1 12 -5 16 5 > 1, swap

1 5 12 -5 16 5 < 12, ok

1 5 12 -5 16 12 > -5, swap

1 5 -5 12 16 12 < 16, ok

1 5 -5 12 16 1 < 5, ok

1 5 -5 12 16 5 > -5, swap

1 -5 5 12 16 5 < 12, ok

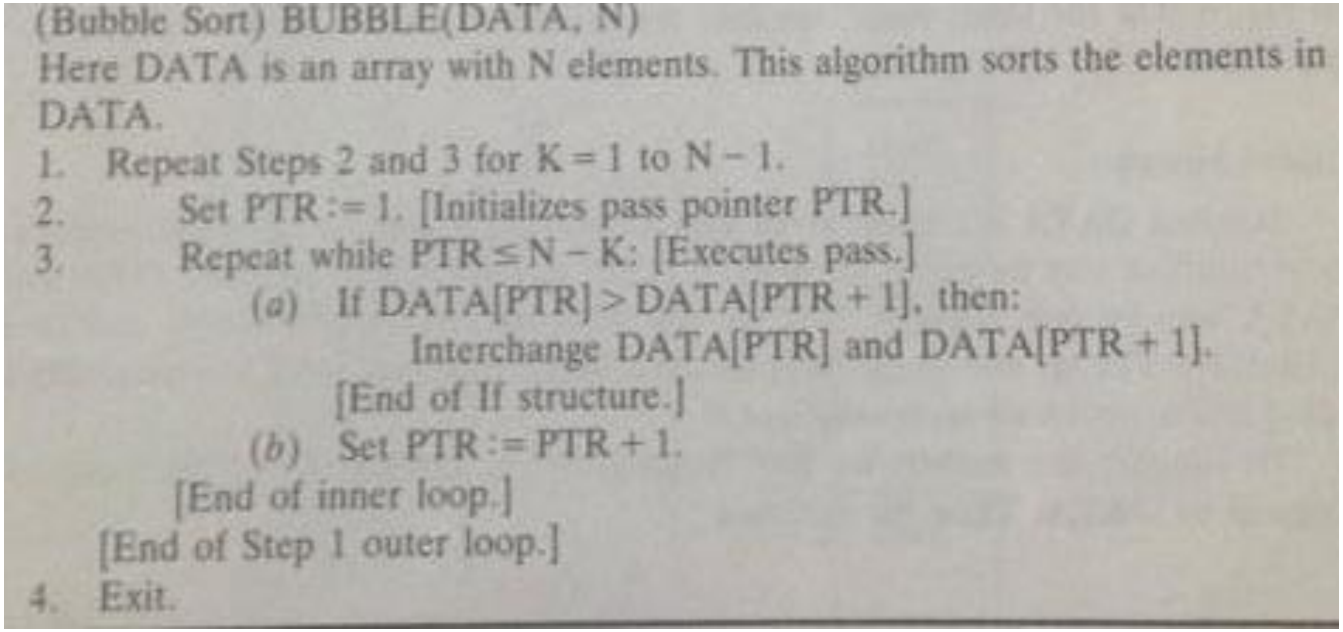
1 -5 5 12 16 1 > -5, swap

-5 1 5 12 16 1 < 5, ok

-5 1 5 12 16 -5 < 1, ok

-5 1 5 12 16 sorted

Sorting; Bubble sort: Algorithm 4.4



(Bubble Sort) BUBBLE(DATA, N)

Here DATA is an array with N elements. This algorithm sorts the elements in DATA.

```
1.  Repeat Steps 2 and 3 for K = 1 to N − 1.
2.        Set PTR := 1. [Initializes pass pointer PTR.]
3.        Repeat while PTR ≤ N − K: [Executes pass.]
              (a)  If DATA[PTR] > DATA[PTR + 1], then:
                        Interchange DATA[PTR] and DATA[PTR + 1].
                   [End of If structure.]
              (b)  Set PTR := PTR + 1.
          [End of inner loop.]
     [End of Step 1 outer loop.]
4.  Exit.
```

Algorithm: (Bubble Sort) BUBBLE (DATA, N)

Here DATA is an Array with N elements. This algorithm sorts the elements in DATA.

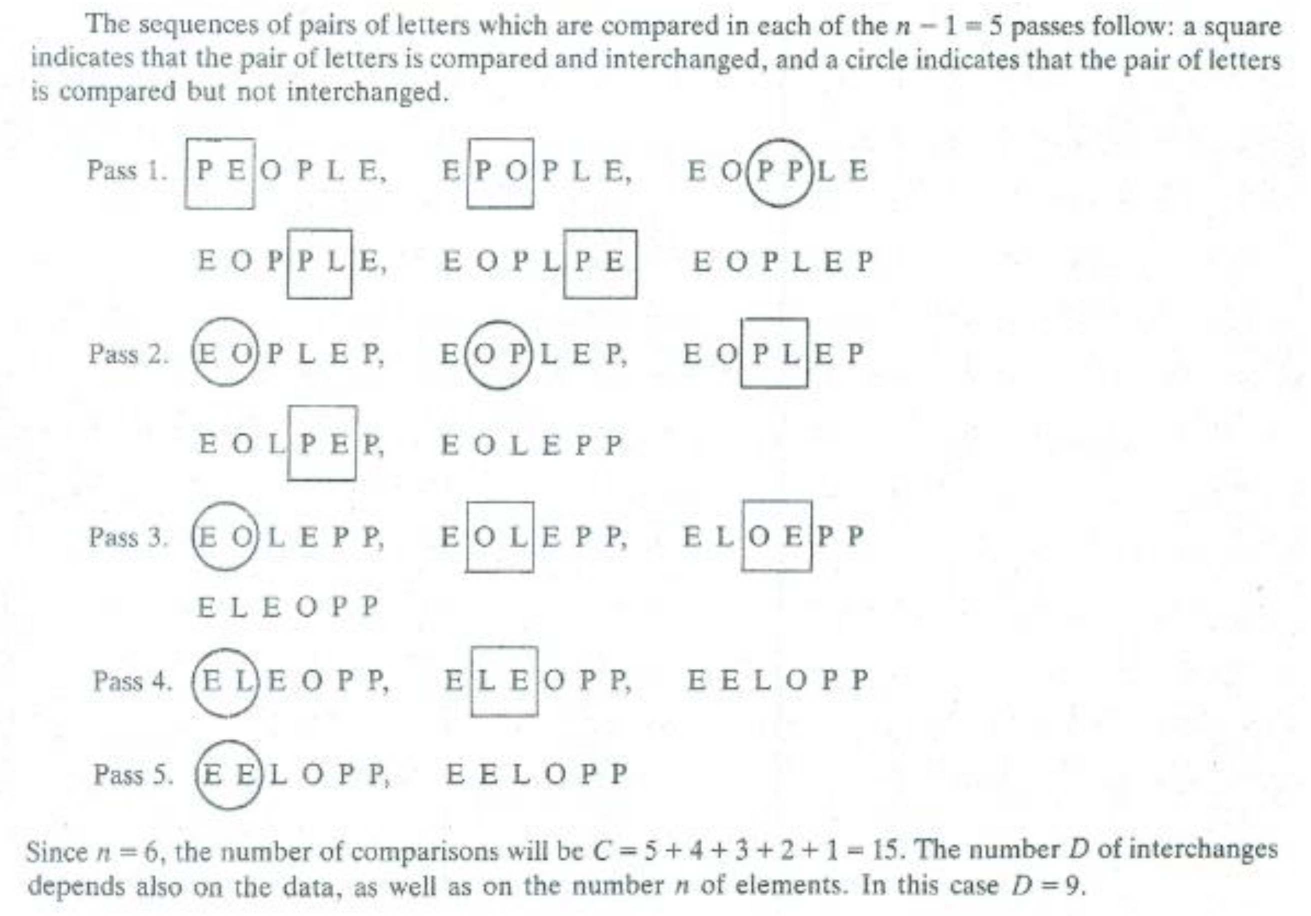
```
1.  for pass=1 to N-1.
2.     for (i=0; i<= N-Pass; i++)
3.         If DATA[i]>DATA[i+1], then:
                          Interchange DATA[i] and DATA[i+1].
                  [End of If Structure.]
             [End of inner loop.]
       [End of Step 1 outer loop.]
4.  Exit.
```

Example

Example. Using the **bubble sort algorithm, Algorithm 4.4, find the number C of comparisons and the number D of interchanges which alphabetize the n =6 letters in PEOPLE.**

Example

Example. Using the **bubble sort algorithm, Algorithm 4.4, find the number C of comparisons and the number D of interchanges which alphabetize the n =6 letters in PEOPLE.**



The sequences of pairs of letters which are compared in each of the $n - 1 = 5$ passes follow: a square indicates that the pair of letters is compared and interchanged, and a circle indicates that the pair of letters is compared but not interchanged.

Pass 1. P E O P L E, E P O P L E, E O P P L E

E O P P L E, E O P L P E E O P L E P

Pass 2. E O P L E P, E O P L E P, E O P L E P

E O L P E P, E O L E P P

Pass 3. E O L E P P, E O L E P P, E L O E P P

E L E O P P

Pass 4. E L E O P P, E L E O P P, E E L O P P

Pass 5. E E L O P P, E E L O P P

Since $n = 6$, the number of comparisons will be $C = 5 + 4 + 3 + 2 + 1 = 15$. The number D of interchanges depends also on the data, as well as on the number n of elements. In this case $D = 9$.

Bubble Sort Time Complexity

- **Best-Case Time Complexity**
  - The scenario under which the algorithm will do the least amount of work (finish the fastest)

- **Worst-Case Time Complexity**
  - The scenario under which the algorithm will do the largest amount of work (finish the slowest).

Bubble Sort Time Complexity

- **Best-Case Time Complexity**
  - **Array is already sorted**
  - **Need 1 iteration with (N-1) comparisons**

Called Linear Time
O(N)
Order-of-N

Called Quadratic Time
$O(N^2)$
Order-of-N-square

- **Worst-Case Time Complexity**
  - **Need N-1 iterations**
  - **$(N-1) + (N-2) + (N-3) + \dots + (1) = (N-1) * N / 2$**

Searching in Linear Array:

- The process of finding a particular element of an array is called Searching”.
- If the item is not present in the array, then the search is unsuccessful.
- There are two types of search (Linear search and Binary Search)

Linear Search:

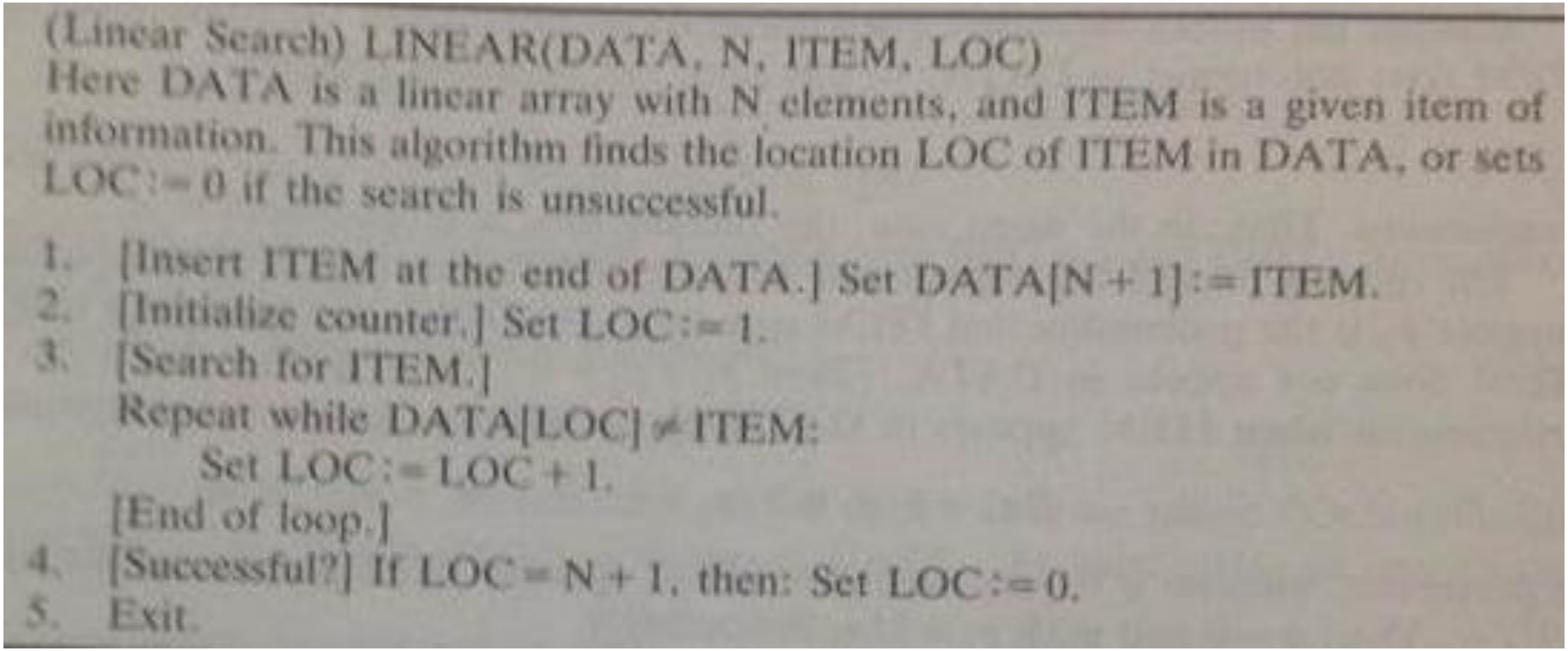
The linear search compares each element of the array with the search key until the search key is found.

To determine that a value is not in the array, the

program must compare the search key to every element in the array.

It is also called “Sequential Search” because it traverses the data sequentially to locate the element.

Linear Array:



(Linear Search) LINEAR(DATA, N, ITEM, LOC)
Here DATA is a linear array with N elements, and ITEM is a given item of information. This algorithm finds the location LOC of ITEM in DATA, or sets LOC := 0 if the search is unsuccessful.

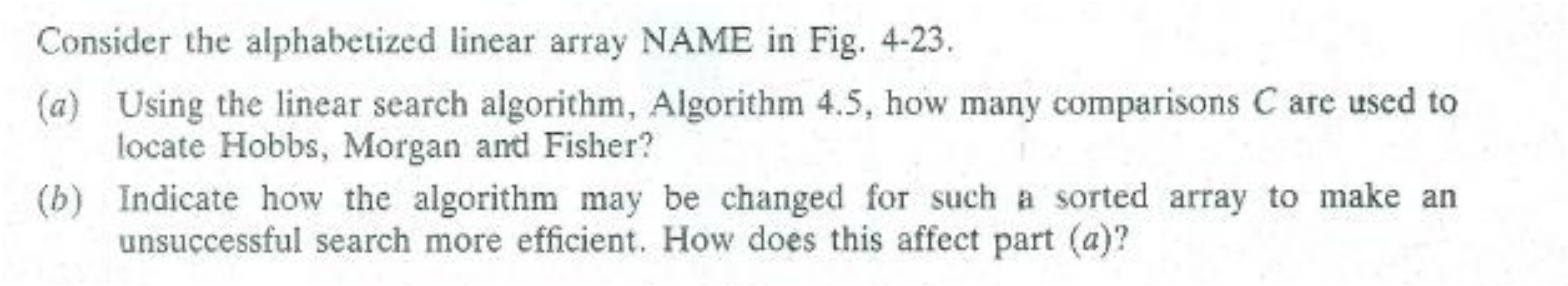
```
1. [Insert ITEM at the end of DATA.] Set DATA[N + 1] := ITEM.
2. [Initialize counter.] Set LOC := 1.
3. [Search for ITEM.]
   Repeat while DATA[LOC] ≠ ITEM:
       Set LOC := LOC + 1.
   [End of loop.]
4. [Successful?] If LOC = N + 1, then: Set LOC := 0.
5. Exit.
```

Linear search Complexity

- **Worst-Case Time Complexity**
  - **Need n+1 iterations**
  - **F(n)=n+1**

- **Average-Case Time Complexity**
  - **F(n)=(n+1)/2**

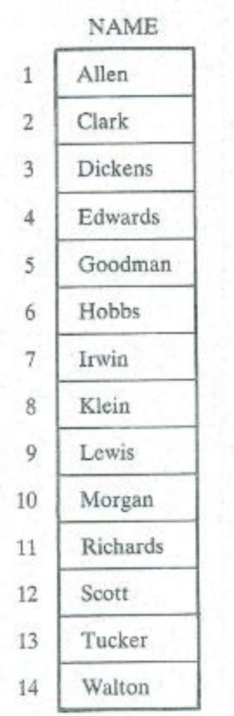
Linear search Complexity



Consider the alphabetized linear array NAME in Fig. 4-23.

(*a*) Using the linear search algorithm, Algorithm 4.5, how many comparisons C are used to locate Hobbs, Morgan and Fisher?

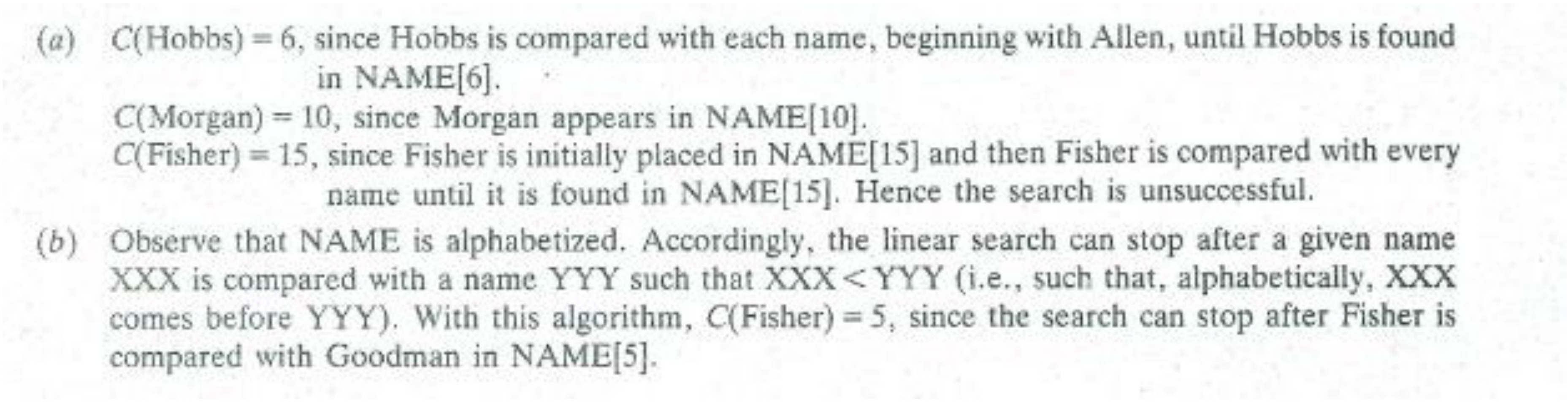
(*b*) Indicate how the algorithm may be changed for such a sorted array to make an unsuccessful search more efficient. How does this affect part (*a*)?



	NAME
1	Allen
2	Clark
3	Dickens
4	Edwards
5	Goodman
6	Hobbs
7	Irwin
8	Klein
9	Lewis
10	Morgan
11	Richards
12	Scott
13	Tucker
14	Walton

Fig. 4-23

Linear search Complexity



(*a*) C(Hobbs) = 6, since Hobbs is compared with each name, beginning with Allen, until Hobbs is found in NAME[6].

C(Morgan) = 10, since Morgan appears in NAME[10].

C(Fisher) = 15, since Fisher is initially placed in NAME[15] and then Fisher is compared with every name until it is found in NAME[15]. Hence the search is unsuccessful.

(*b*) Observe that NAME is alphabetized. Accordingly, the linear search can stop after a given name XXX is compared with a name YYY such that XXX < YYY (i.e., such that, alphabetically, XXX comes before YYY). With this algorithm, C(Fisher) = 5, since the search can stop after Fisher is compared with Goodman in NAME[5].

Chapter 4: Arrays, Records and Pointers

4.9. Multidimensional Arrays

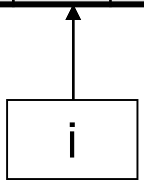
1. Two-Dimensional Arrays
2. Example 4.11
3. Representation of Two-Dimensional Arrays in Memory
4. Example 4.12

Binary search

Sequential search

- **sequential search**: Locates a target value in an array / list by examining each element from start to finish.
  - How many elements will it need to examine?
  - Example: Searching the array below for the value **42**:

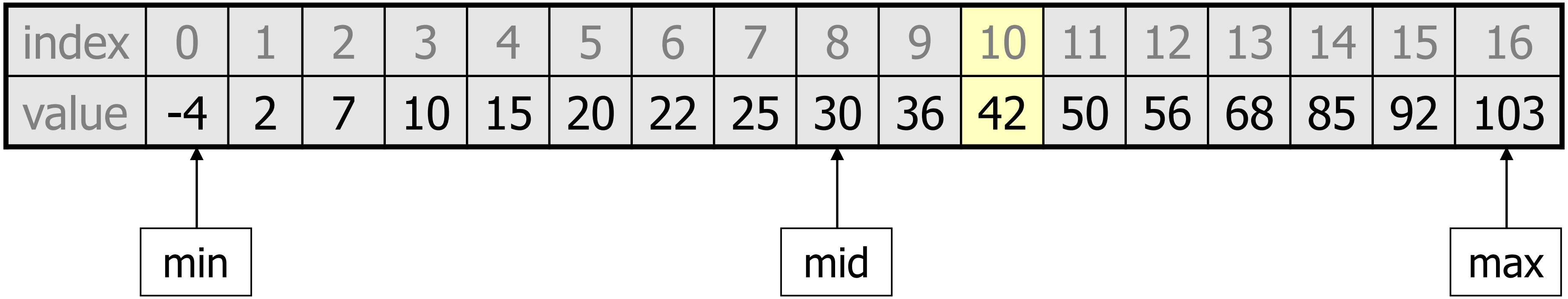
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103



  - Notice that the array is sorted. Could we take advantage of this?

Binary search

- **binary search**: Locates a target value in a *sorted* array / list by successively eliminating half of the array from consideration.
  - How many elements will it need to examine?
  - Example: Searching the array below for the value **42**:



index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

Runtime Efficiency

- How much better is binary search than sequential search?

- **efficiency**: A measure of the use of computing resources by code.
  - can be relative to speed (time), memory (space), etc.
  - most commonly refers to run time

- Assume the following:
  - Any single C# statement takes the same amount of time to run.
  - A method call's runtime is measured by the total of the statements inside the method's body.
  - A loop's runtime, if the loop repeats N times, is N times the runtime of the statements in its body.

Sequential search

- What is its complexity class?

```
public int indexOf(int value) {
    for (int i = 0; i < size; i++) {
        if (elementData[i] == value) {
            return i;
        }
    }
    return -1;    // not found
}
```

N

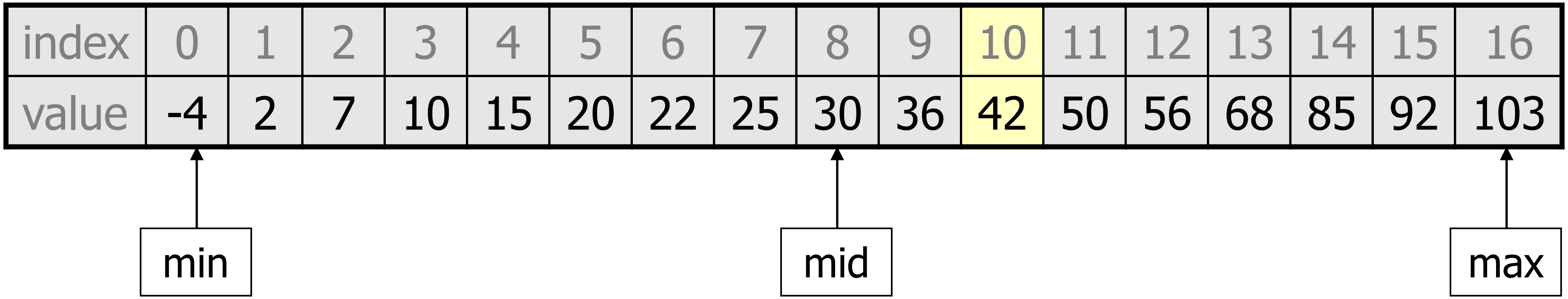
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

- On average, "only" N/2 elements are visited
  - 1/2 is a constant that can be ignored → O(N)

Binary search

- **binary search** successively eliminates half of the elements.
  - *Algorithm:* Examine the middle element of the array.
    - If it is too big, eliminate the right half of the array and repeat.
    - If it is too small, eliminate the left half of the array and repeat.
    - Else it is the value we're searching for, so stop.

  - Which indexes does the algorithm examine to find value **42**?
  - What is the runtime complexity class of binary search?



index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

Binary search runtime

- For an array of size N, it eliminates ½ until 1 element remains.

  N, N/2, N/4, N/8, ..., 4, 2, 1

  - How many divisions does it take?

- Think of it from the other direction:
  - How many times do I have to multiply by 2 to reach N?

    1, 2, 4, 8, ..., N/4, N/2, N
  - Call this number of multiplications "x".

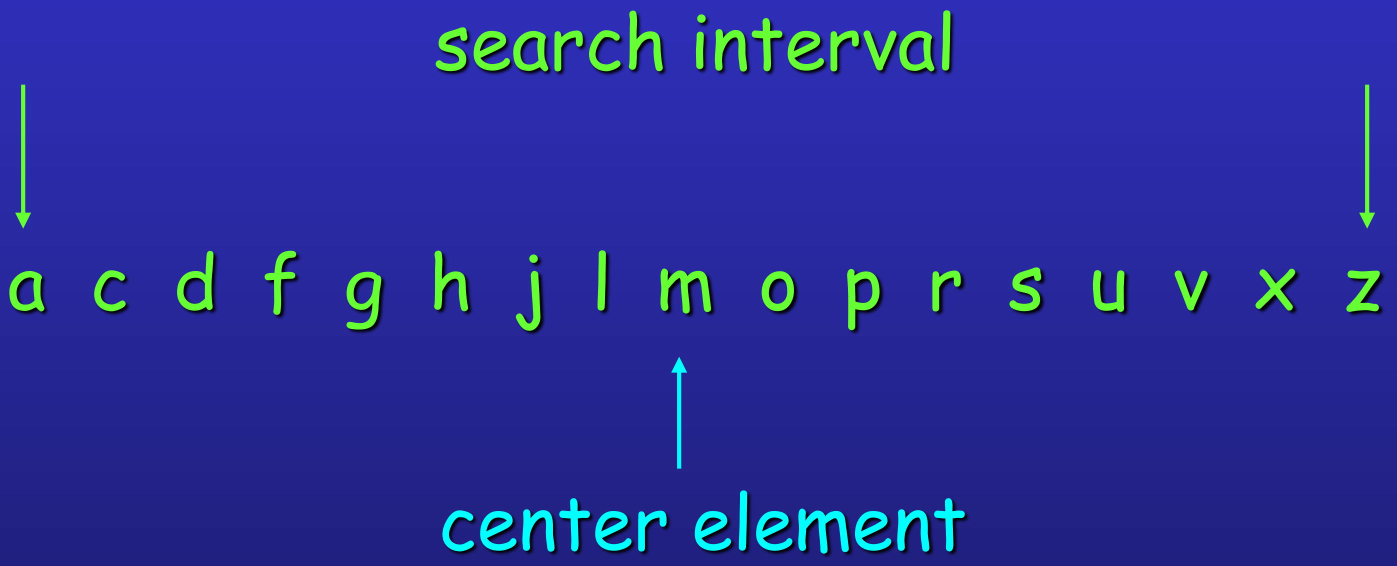
$2^x = N$

$\mathbf{x = \log_2 N}$

- Binary search is in the **logarithmic** complexity class.

Algorithm Examples

binary search for the letter ‘j’



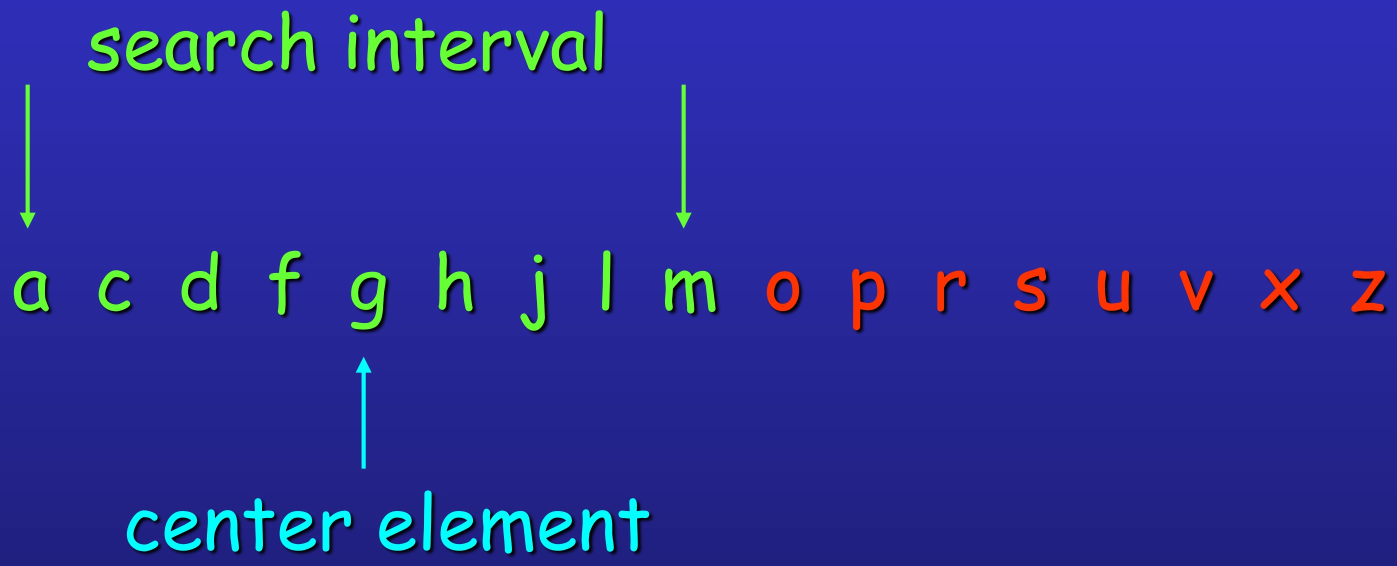
search interval

a c d f g h j l m o p r s u v x z

center element

Algorithm Examples

binary search for the letter 'j'



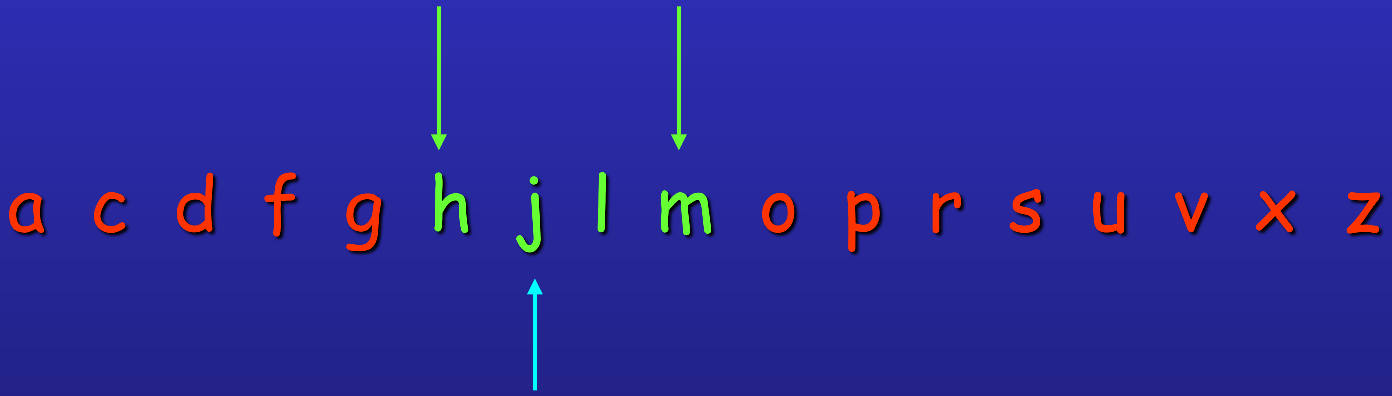
search interval

a c d f g h j l m o p r s u v x z

center element

Algorithm Examples

binary search for the letter 'j'



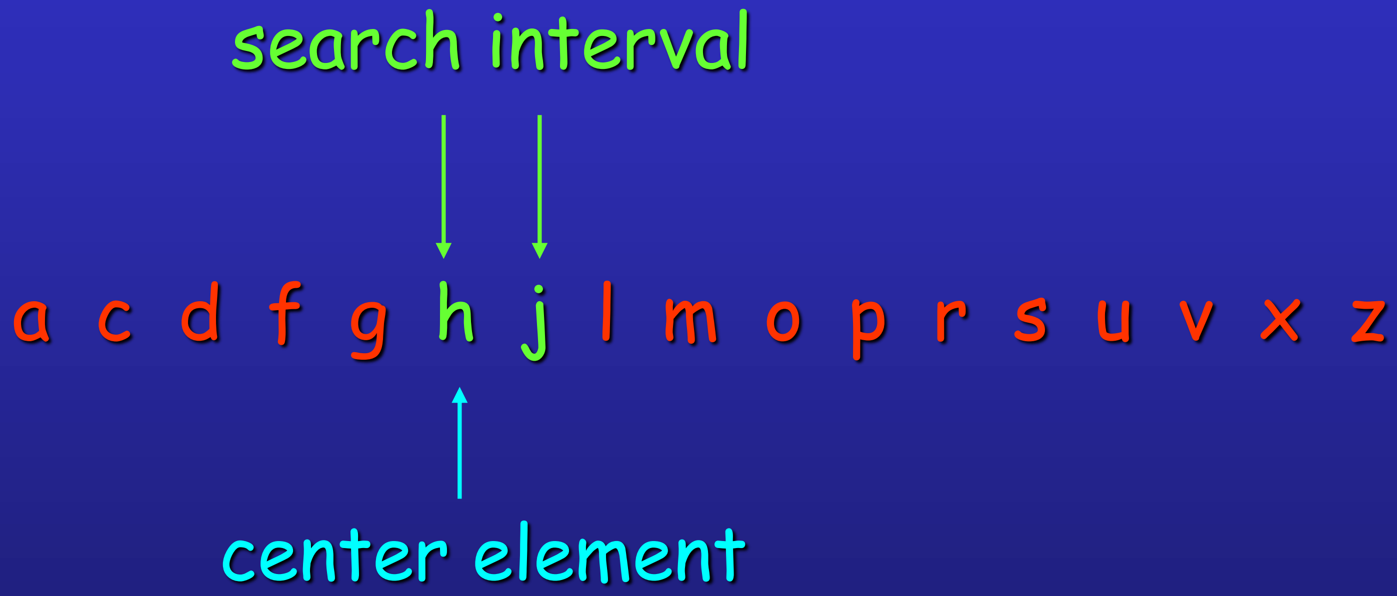
search interval

a c d f g h j l m o p r s u v x z

center element

Algorithm Examples

binary search for the letter 'j'



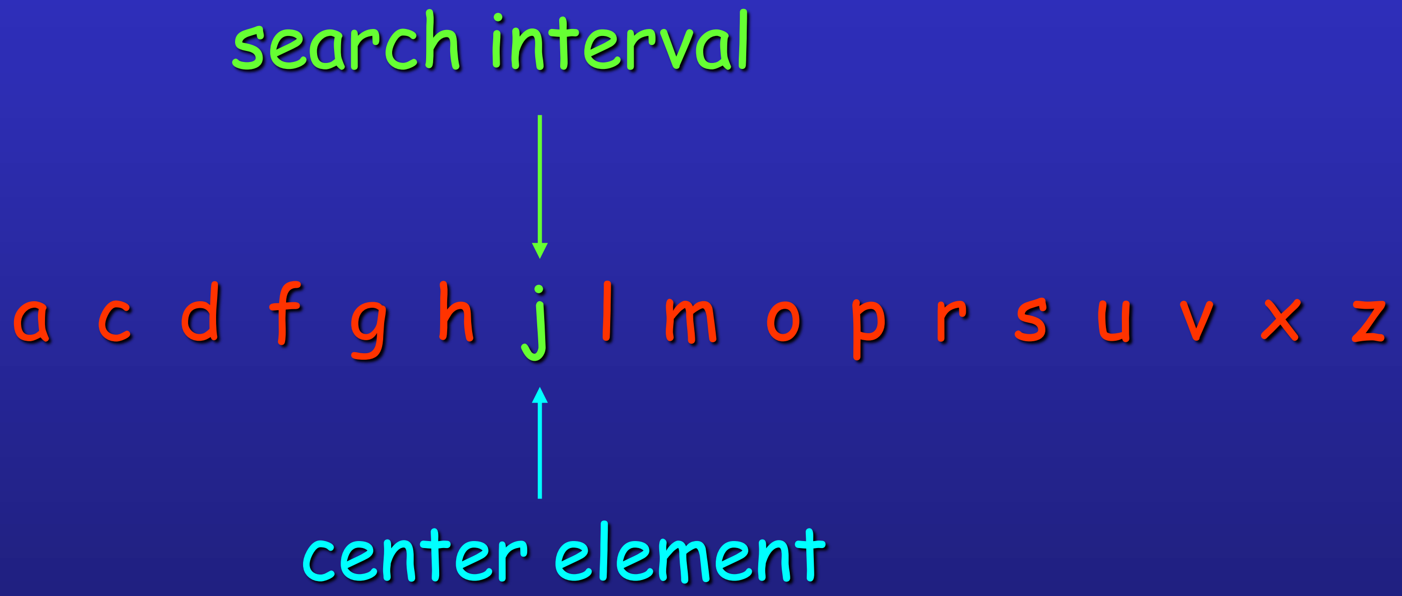
search interval

a c d f g h j l m o p r s u v x z

center element

Algorithm Examples

binary search for the letter 'j'



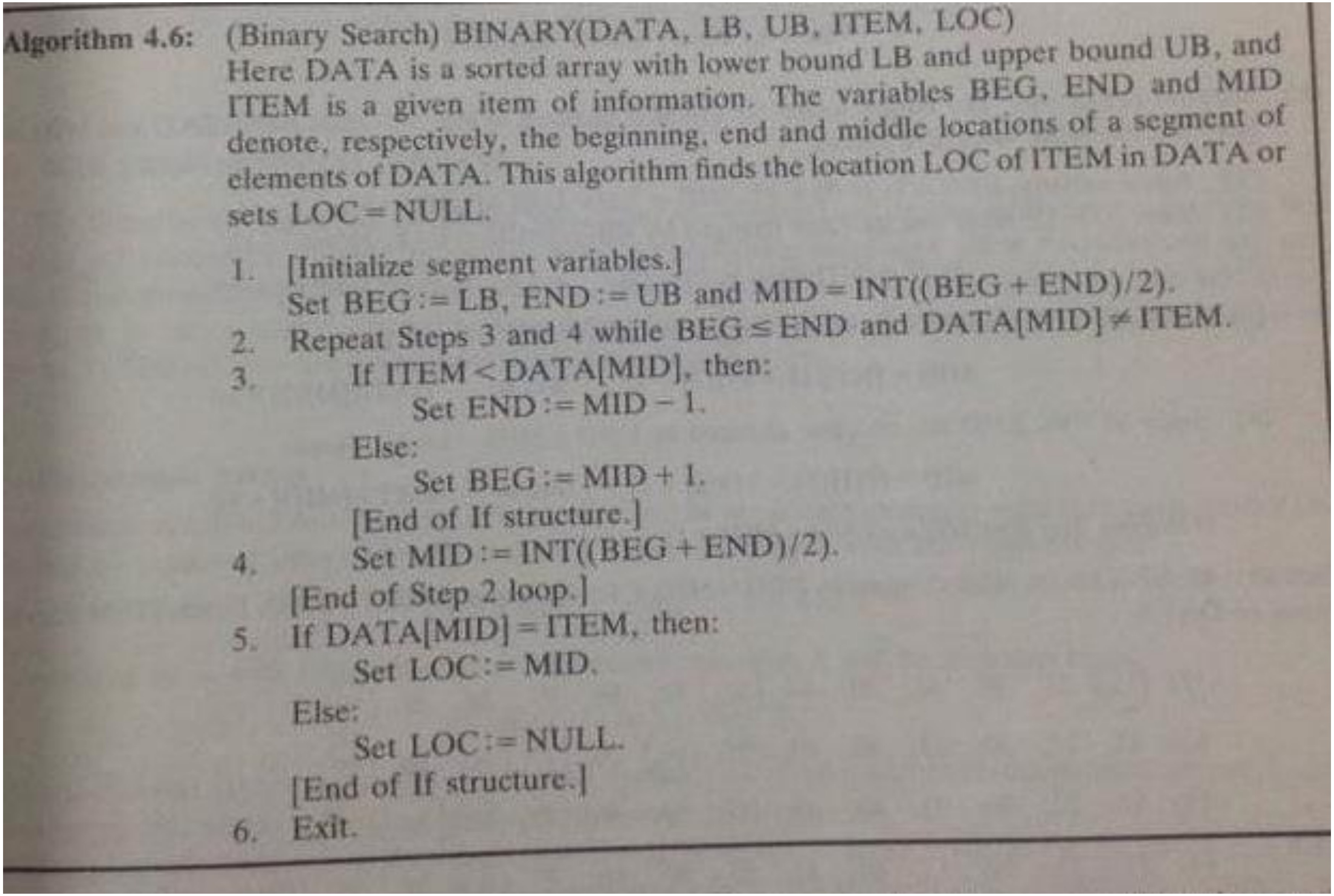
search interval

a c d f g h j l m o p r s u v x z

center element

found !

Binary search

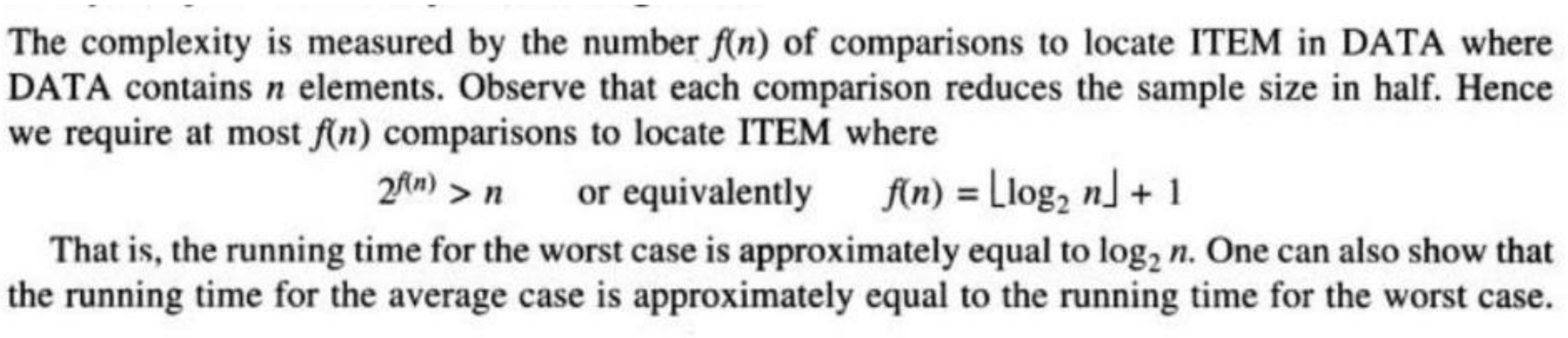


Algorithm 4.6: (Binary Search) BINARY(DATA, LB, UB, ITEM, LOC)
Here DATA is a sorted array with lower bound LB and upper bound UB, and ITEM is a given item of information. The variables BEG, END and MID denote, respectively, the beginning, end and middle locations of a segment of elements of DATA. This algorithm finds the location LOC of ITEM in DATA or sets LOC = NULL.

```
1.  [Initialize segment variables.]
    Set BEG := LB, END := UB and MID = INT((BEG + END)/2).
2.  Repeat Steps 3 and 4 while BEG ≤ END and DATA[MID] ≠ ITEM.
3.          If ITEM < DATA[MID], then:
                Set END := MID − 1.
            Else:
                Set BEG := MID + 1.
            [End of If structure.]
4.          Set MID := INT((BEG + END)/2).
    [End of Step 2 loop.]
5.  If DATA[MID] = ITEM, then:
            Set LOC := MID.
    Else:
            Set LOC := NULL.
    [End of If structure.]
6.  Exit.
```

Complexity of the Binary search algorithm

- **Worst-Case Time Complexity**
  - **F(n)=log(n)+1**

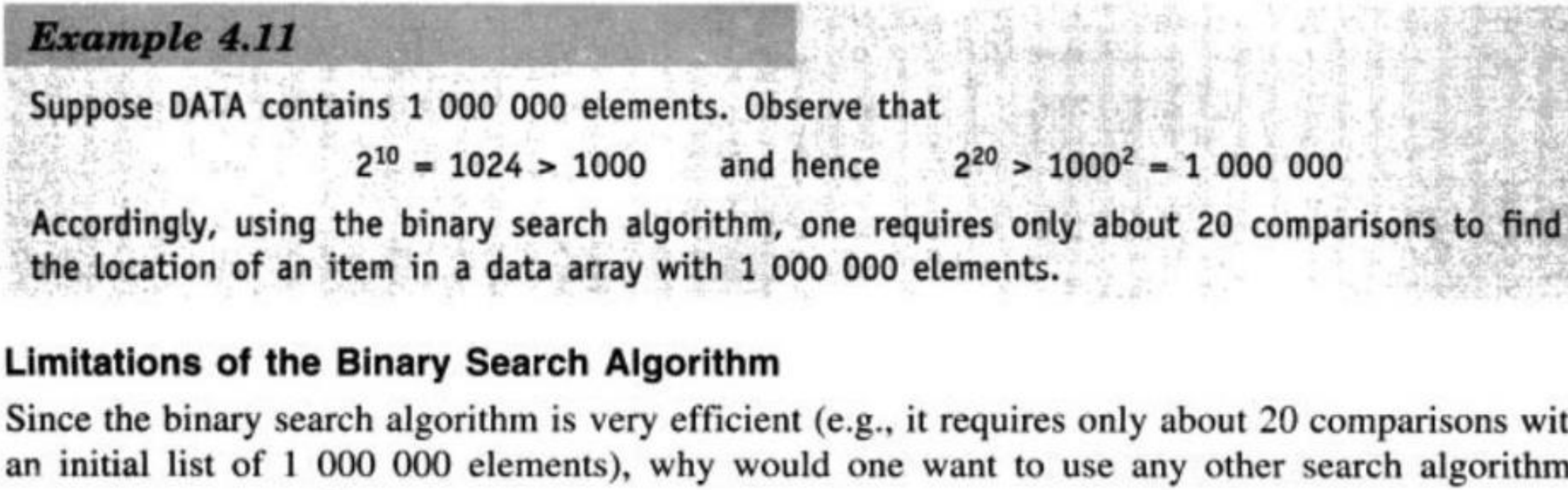


The complexity is measured by the number $f(n)$ of comparisons to locate ITEM in DATA where DATA contains n elements. Observe that each comparison reduces the sample size in half. Hence we require at most $f(n)$ comparisons to locate ITEM where

$$2^{f(n)} > n \qquad \text{or equivalently} \qquad f(n) = \lfloor \log_2 n \rfloor + 1$$

That is, the running time for the worst case is approximately equal to $\log_2 n$. One can also show that the running time for the average case is approximately equal to the running time for the worst case.

Example



Example 4.11

Suppose DATA contains 1 000 000 elements. Observe that

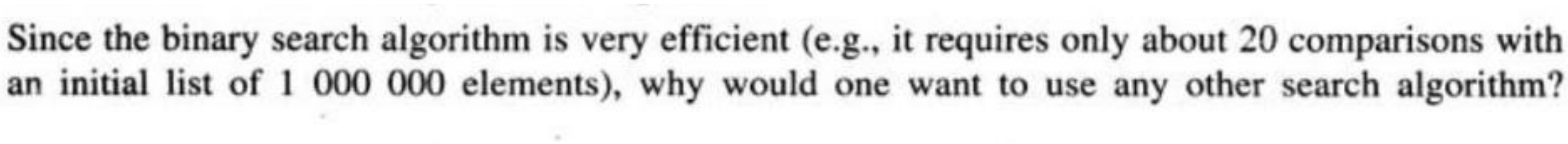
$$2^{10} = 1024 > 1000 \quad \text{and hence} \quad 2^{20} > 1000^2 = 1\,000\,000$$

Accordingly, using the binary search algorithm, one requires only about 20 comparisons to find the location of an item in a data array with 1 000 000 elements.

Limitations of the Binary Search Algorithm

Since the binary search algorithm is very efficient (e.g., it requires only about 20 comparisons wit an initial list of 1 000 000 elements), why would one want to use any other search algorithm

Limitations of the binary Search algorithm



Since the binary search algorithm is very efficient (e.g., it requires only about 20 comparisons with an initial list of 1 000 000 elements), why would one want to use any other search algorithm?

4.9 Multidimensional Arrays

Multidimensional Arrays

1. Two-Dimensional Arrays
2. Example 4.11
3. Representation of Two-Dimensional Arrays in Memory
4. Example 4.12

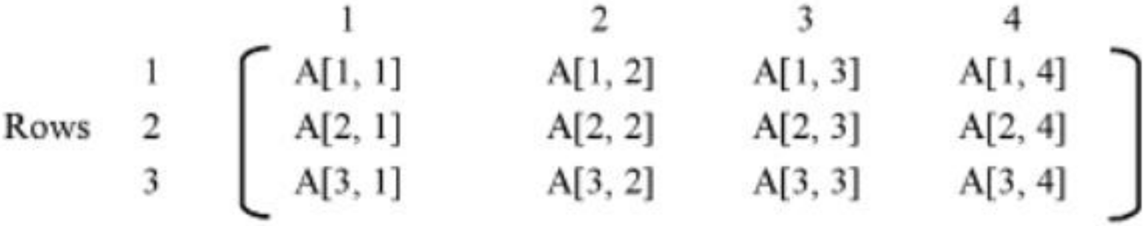
Multi-dimensional Arrays

- Linear array is one dimensional array,
  - use one subscript such as → A[i]
- Two dimensional array
  - uses two subscripts such as → A[i,j]
- Multidimensional array
  - uses 3-7 subscripts such as → A[i,j,k].

Two Dimensional Arrays

Two dimensional Array:

- A is a collection of mxn data elements such that each element is specified by a pair of integers (such as J, K) called subscripts with the property that 1≤ J≤ M and 1≤ K≤n
- The element of A with first subscript J and second subscript K will be denoted by A [J,K].
- Two dimensional arrays are sometimes called (matrices) matrix array.



$$\begin{array}{ll} & \begin{array}{cccc} \quad 1 & \qquad 2 & \qquad 3 & \qquad 4 \end{array} \\ \text{Rows} \begin{array}{c} 1 \\ 2 \\ 3 \end{array} & \left[\begin{array}{cccc} A[1,1] & A[1,2] & A[1,3] & A[1,4] \\ A[2,1] & A[2,2] & A[2,3] & A[2,4] \\ A[3,1] & A[3,2] & A[3,3] & A[3,4] \end{array}\right] \end{array}$$

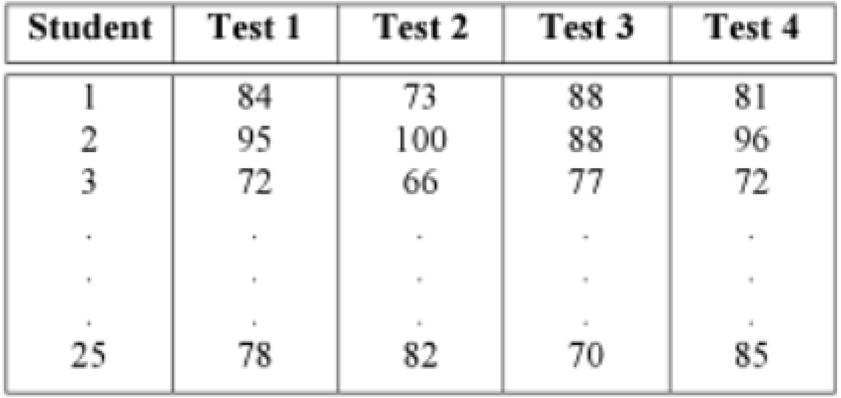
Two Dimensional Arrays

- It is also called matrices in mathematics and table in business applications.
- Size is m.n
- Length=upper bound –lower bound+1
- LB of Regular arrays=1
- Dimensions of INTEGER NUMB(2:5,-3:1)
  - length of first dimension =?
  - length of second dimension= ?
  - The NUMB dimension=?

Example 4.11

Example 4.11

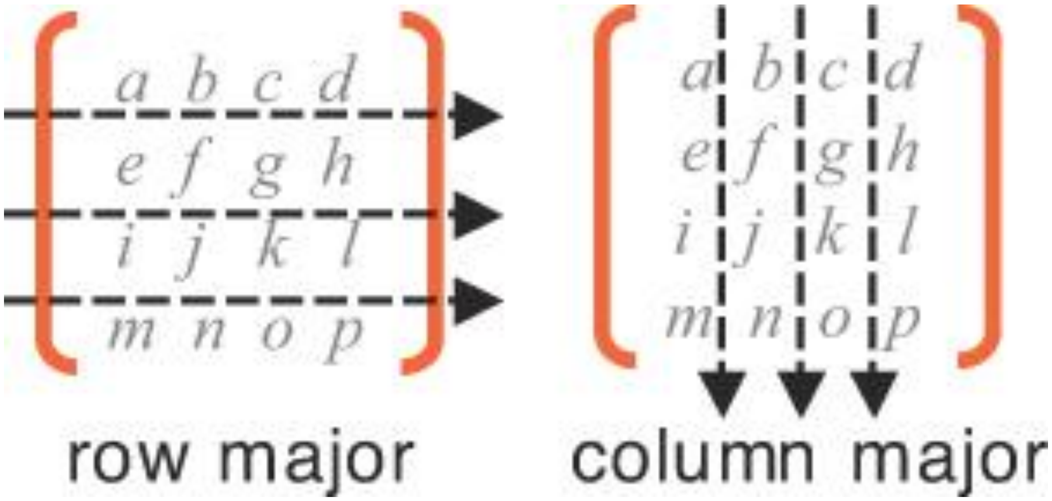
- Class of 25 students is given 4 tests.
- Store the data in a 25×4 matrix SCORE
- SCORE[K,L] contains the Kth student's score on the Lth Test.



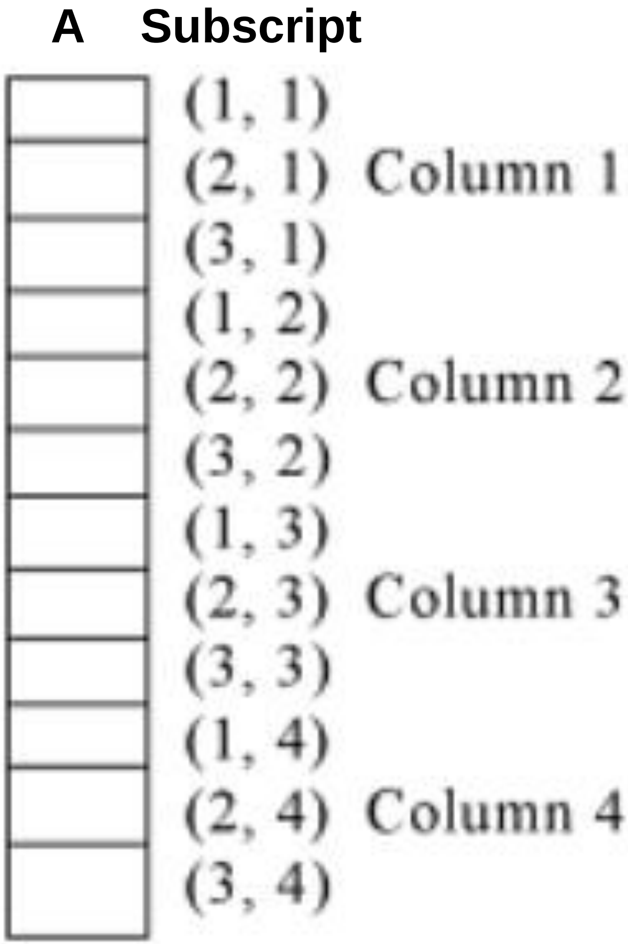
Student	Test 1	Test 2	Test 3	Test 4
1	84	73	88	81
2	95	100	88	96
3	72	66	77	72
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
25	78	82	70	85

Representation of Two-Dimensional Arrays in Memory

- M×N rectangular matrix will be represented in memory by a block of m.n sequential memory locations.
- Programing language will store the array A either in two ways:
  1. Column Major Order:
  2. Row Major Order
- Representation depends upon the program not user



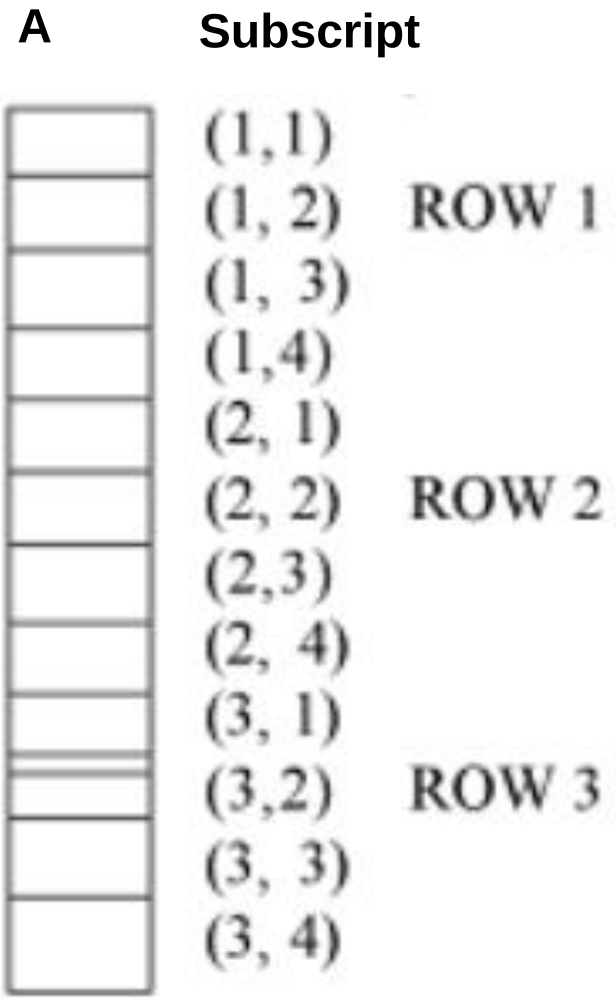
Representation of Two-Dimensional Arrays in Memory



A **Subscript**

(1, 1)
(2, 1) Column 1
(3, 1)
(1, 2)
(2, 2) Column 2
(3, 2)
(1, 3)
(2, 3) Column 3
(3, 3)
(1, 4)
(2, 4) Column 4
(3, 4)

(a) Column-major order

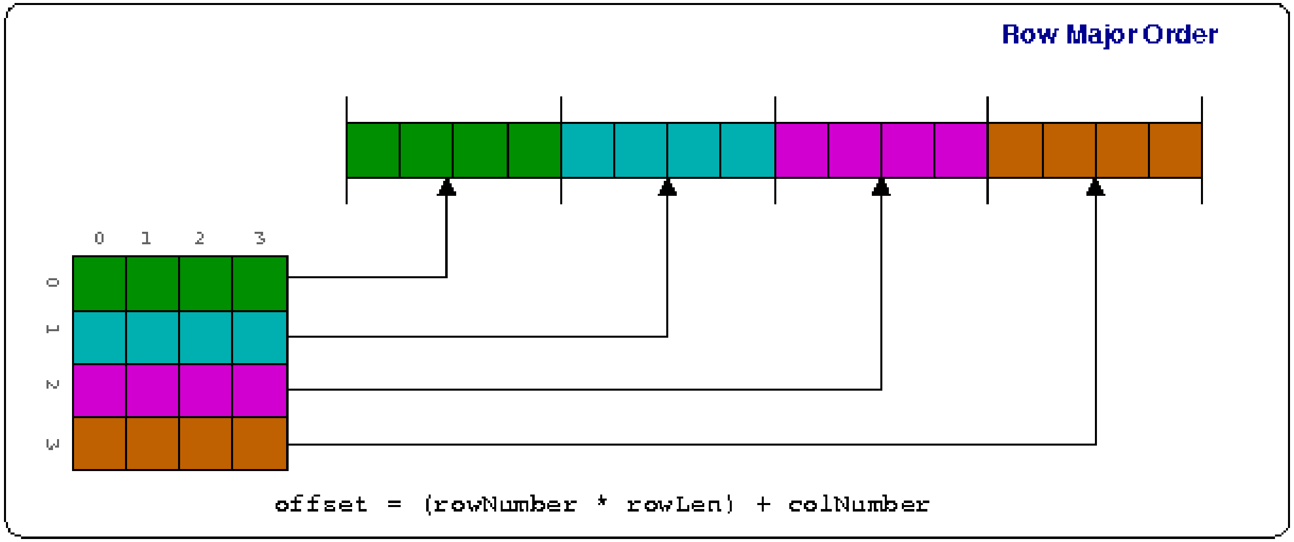


A **Subscript**

(1,1)
(1, 2) ROW 1
(1, 3)
(1,4)
(2, 1)
(2, 2) ROW 2
(2,3)
(2, 4)
(3, 1)
(3,2) ROW 3
(3, 3)
(3, 4)

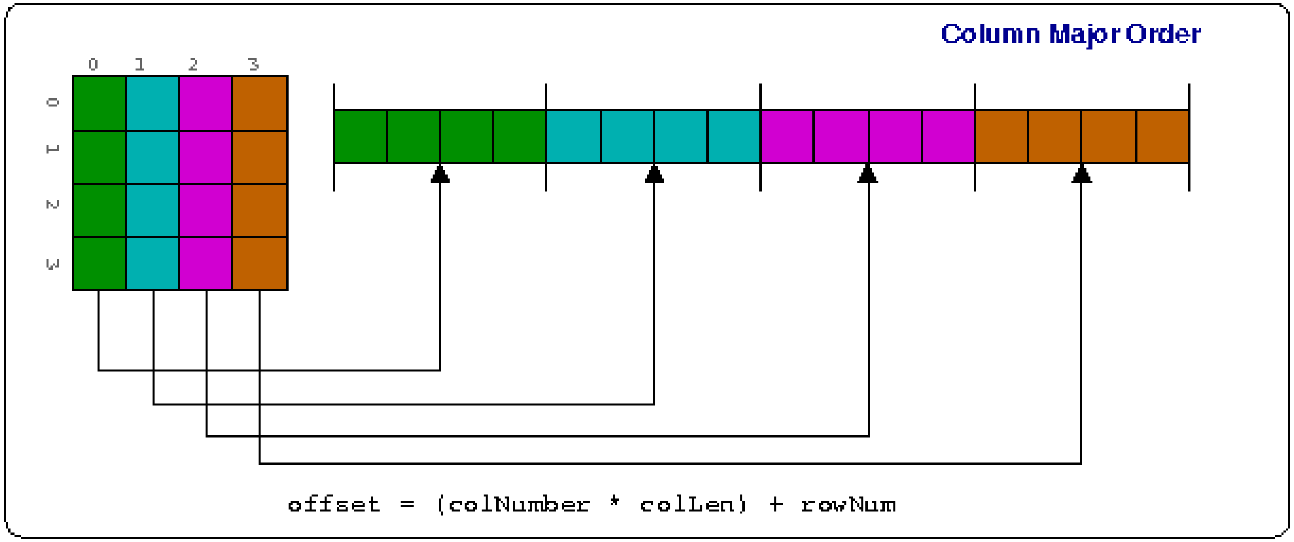
(b) Row-major order

Representation of Two-Dimensional Arrays in Memory



Row Major Order

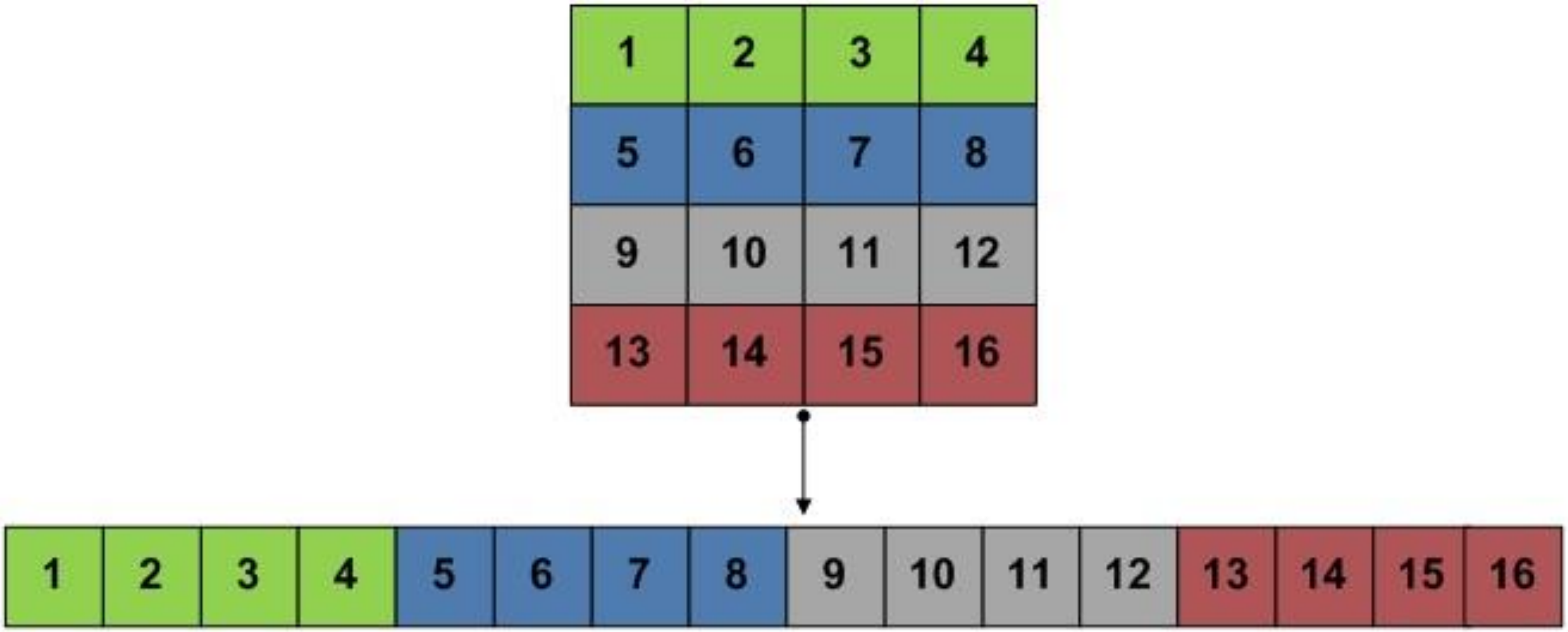
`offset = (rowNumber * rowLen) + colNumber`



Column Major Order

`offset = (colNumber * colLen) + rowNum`

Representation of Two-Dimensional Arrays in Memory



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Representation of Two-Dimensional Arrays in Memory

- Linear array do not keep track of the address LOC(A[k]) of every element A[k],
- but does keep the track of Base(A), the address of first element.
- formula LOC(A[k] = base (A) + w(k-1)
- Same situation holds for **two-dimensional** mxn array A.
- Computer does not keep the Address of all elements in the array
- Computer keeps track of BASE(A) which is the address of the first element A[1,1] of A
- Computer computes the address LOC(A[J,K]) of the element A[J,K] using two different formulas.

In case of Column Major Order:

LOC (A [J, K]) = Base (A) + w [M (K-Col_LB) + (J-Row_LB)]

LOC (A [J, K]) = Base (A) + w [M (K-1) + (J-1)]

- LOC (A [J, K]):is the location of the element in the Jth row and Kth column.
- Base (A) : is the base address of the array A.
- w :is the number of bytes required to store single element of the array A.
- M :is the total number of rows in the array.
- J :is the row number of the element.
- K :is the column number of the element.

In case of Row Major Order:

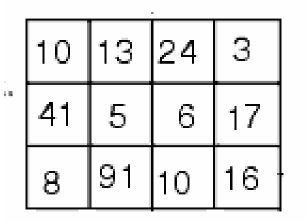
$$\mathrm{LOC}\ (A\ [J, K]) = \mathrm{Base}\ (A) + w\ [N\ (J-\mathrm{Row\_LB}) + (K-\mathrm{Col\_LB})]$$

$$\mathrm{LOC}\ (A\ [J, K]) = \mathrm{Base}\ (A) + w\ [N\ (J-1) + (K-1)]$$

- LOC (A [J, K]):is the location of the element in the Jth row and Kth column.
- Base (A) : is the base address of the array A.
- w :is the number of bytes required to store single element of the array A.
- N : is the total number of columns in the array.
- J :is the row number of the element.
- K :is the column number of the element.

Example 4.12

Consider the 3×4 array A



10	13	24	3
41	5	6	17
8	91	10	16

Suppose Base(A)=100 and there are w=4 words per memory cell,

A. suppose the programing language stores two-dimensional arrays using row-major order. Find the location of the element A[2,3]

B. suppose the programing language stores two-dimensional arrays using column-major order. Find the location of the element A[2,3]

Example 4.12

The formula for LOC (A [J, K]) is

LOC (A [J, K]) = Base (A) + w [N (J-Row_LB) + (K-Col_LB)]

Row_LB=1, K-Col_LB=1

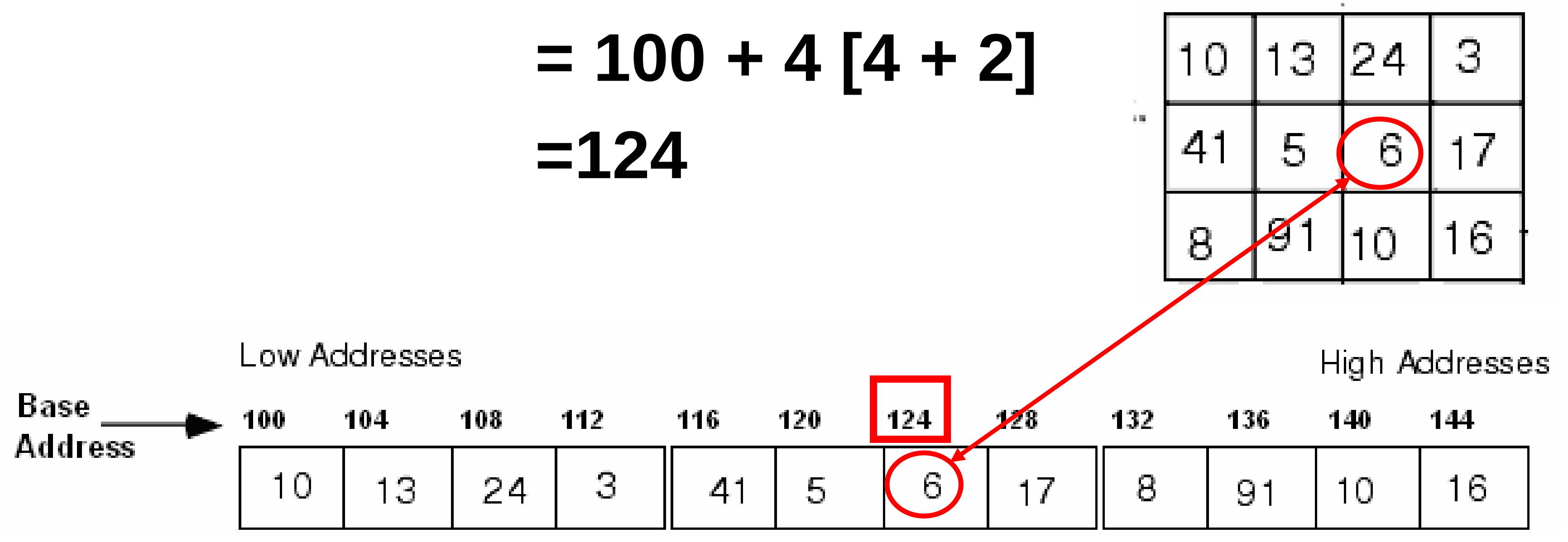
➔ LOC (A [J, K]) = Base (A) + w [N (J-1) + (K-1)]

➢ LOC (A [2, 3]) = 100 + 4 [4 (2-1) + (3-1)]

= 100 + 4 [4 (1) + 2]

= 100 + 4 [4 + 2]

=124



10	13	24	3
41	5	6	17
8	91	10	16

100	104	108	112	116	120	124	128	132	136	140	144
10	13	24	3	41	5	6	17	8	91	10	16

Example 4.12 : Column-major order

The formula for LOC (A [J, K]) is

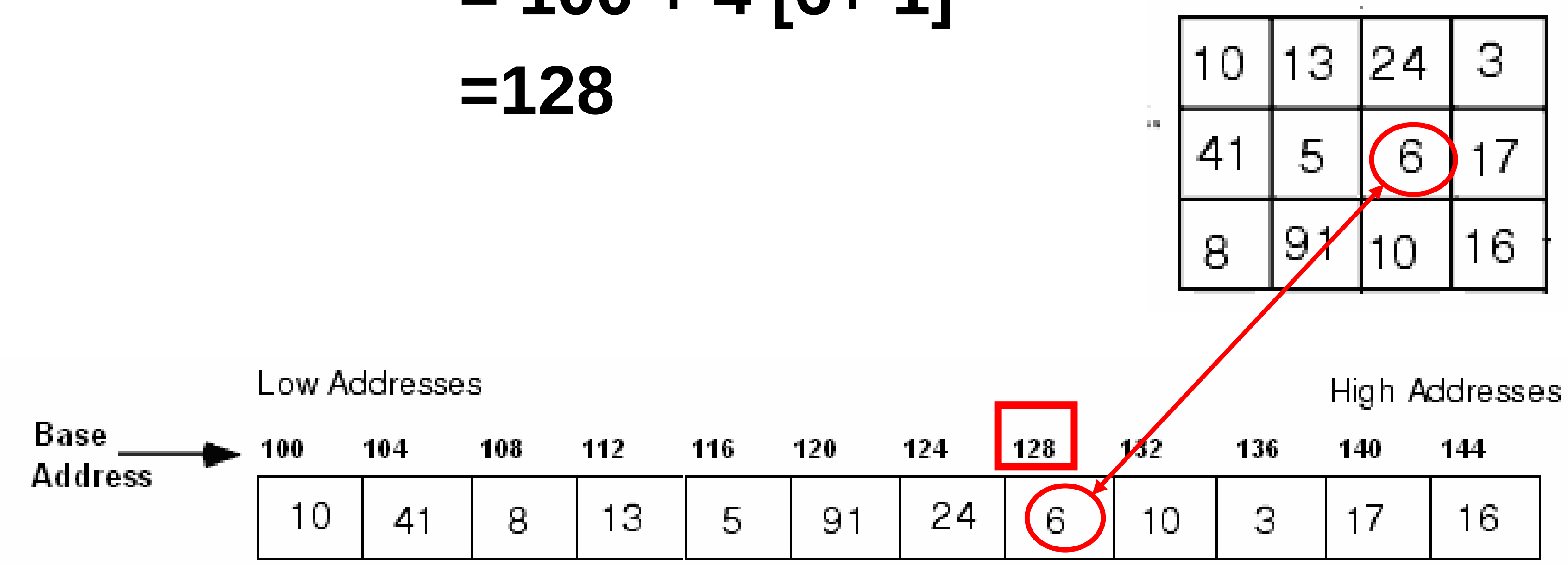
LOC (A [J, K]) = Base (A) + w [M (K-1) + (J-1)]

LOC (A [2, 3]) = 100 + 4 [3 (3-1) + (2-1)]

= 100 + 4 [3 (2) + 1]

= 100 + 4 [6+ 1]

=128

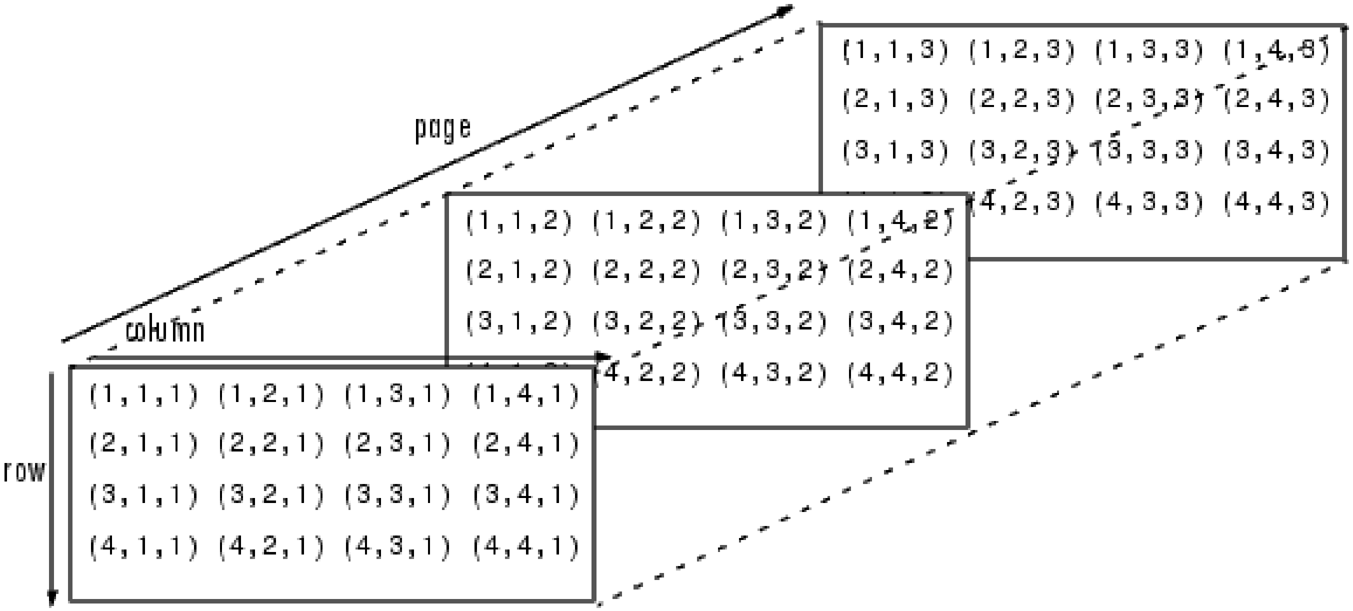


Logical and Physical view

- The difference between the logical and physical view of data
- Logical views of 3×4 matrix array A
- Is rectangular array of data where A[K,J] is an element appears in row J and column K.
- Physical view is the representation in the memory as a linear collection of memory cells
- E.g. certain data may be viewed logically as trees or graphs although physically the data will be stored linearly in memory cells

Multi-dimensional arrays

- The first references array dimension 1, the row.
- The second references dimension 2, the column.
- The third references dimension 3. This illustration uses the concept of a *page* to represent dimensions 3 and higher.



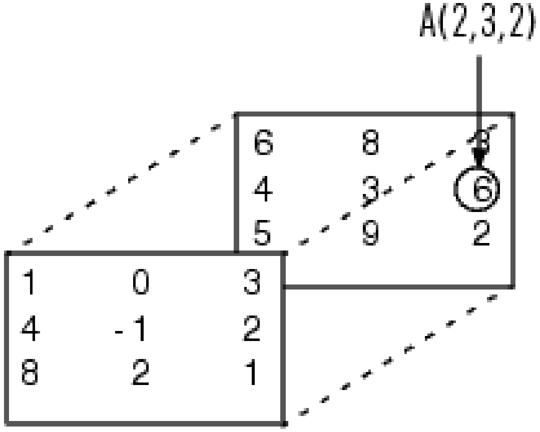
(1,1,3)	(1,2,3)	(1,3,3)	(1,4,3)
(2,1,3)	(2,2,3)	(2,3,3)	(2,4,3)
(3,1,3)	(3,2,3)	(3,3,3)	(3,4,3)
	(4,2,3)	(4,3,3)	(4,4,3)

(1,1,2)	(1,2,2)	(1,3,2)	(1,4,2)
(2,1,2)	(2,2,2)	(2,3,2)	(2,4,2)
(3,1,2)	(3,2,2)	(3,3,2)	(3,4,2)
	(4,2,2)	(4,3,2)	(4,4,2)

(1,1,1)	(1,2,1)	(1,3,1)	(1,4,1)
(2,1,1)	(2,2,1)	(2,3,1)	(2,4,1)
(3,1,1)	(3,2,1)	(3,3,1)	(3,4,1)
(4,1,1)	(4,2,1)	(4,3,1)	(4,4,1)

Multi-dimensional arrays

To access the element in the second row, third column of page 2, for example, you use the subscripts `(2,3,2)`.



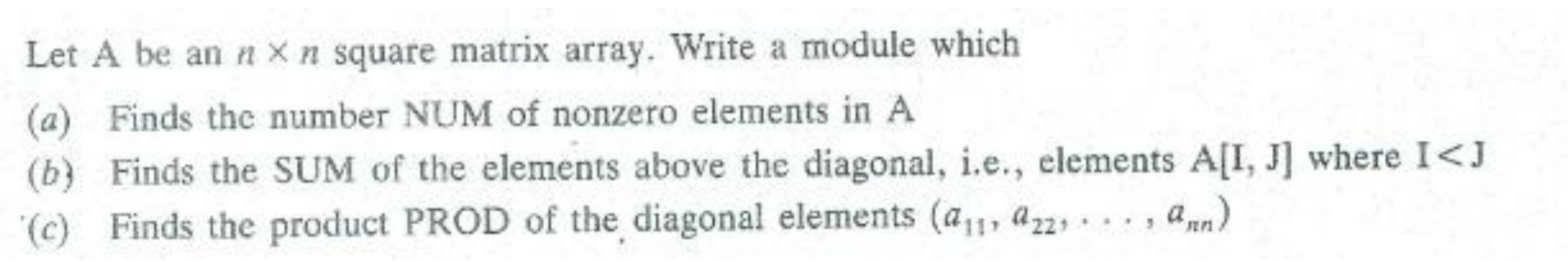
```
A(:,:,1) =

     1     0     3
     4    -1     2
     8     2     1


A(:,:,2) =

     6     8     3
     4     3     6
     5     9     2
```

Example



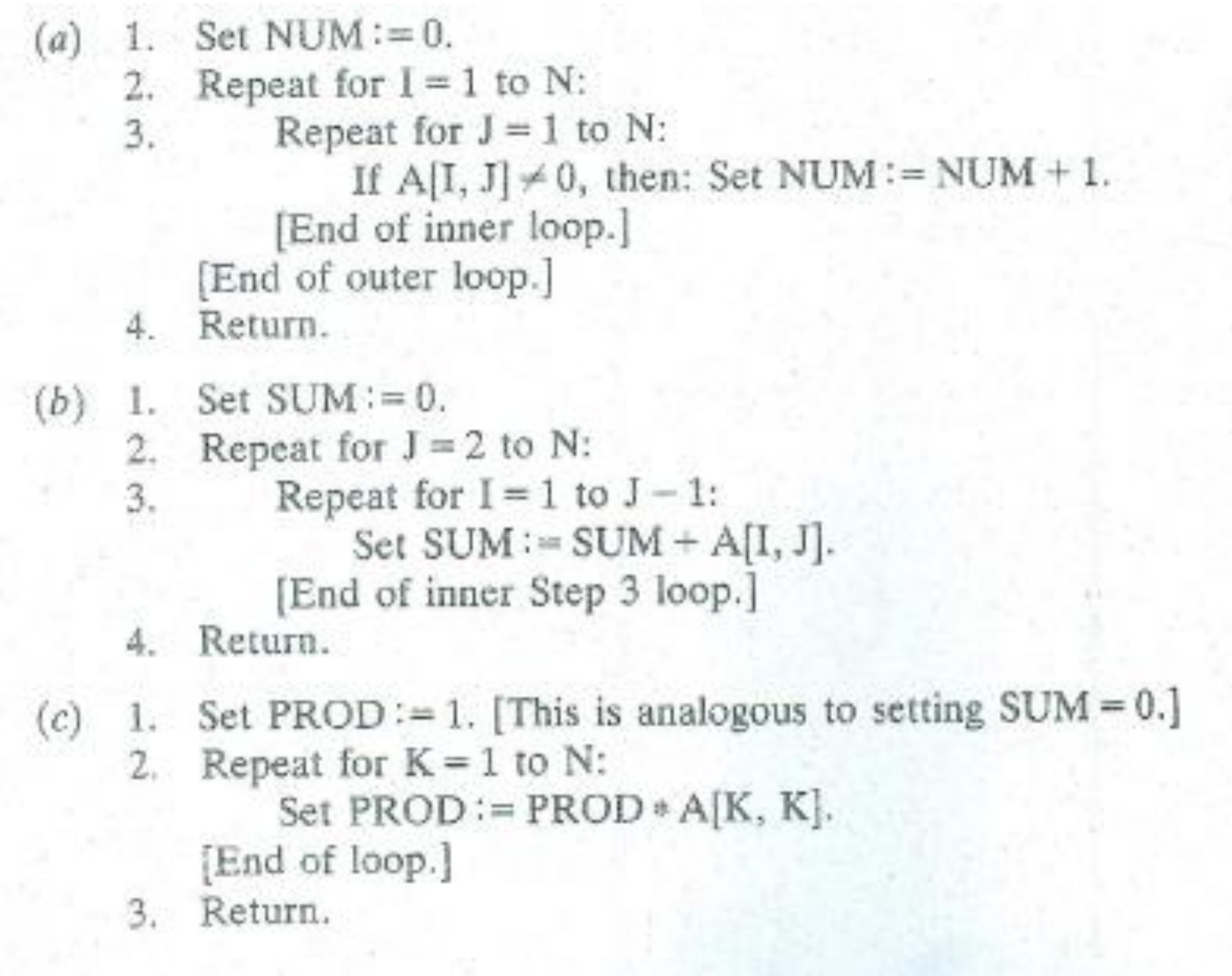
Let A be an $n \times n$ square matrix array. Write a module which

(*a*) Finds the number NUM of nonzero elements in A

(*b*) Finds the SUM of the elements above the diagonal, i.e., elements A[I, J] where I < J

(*c*) Finds the product PROD of the diagonal elements $(a_{11}, a_{22}, \ldots, a_{nn})$

Example

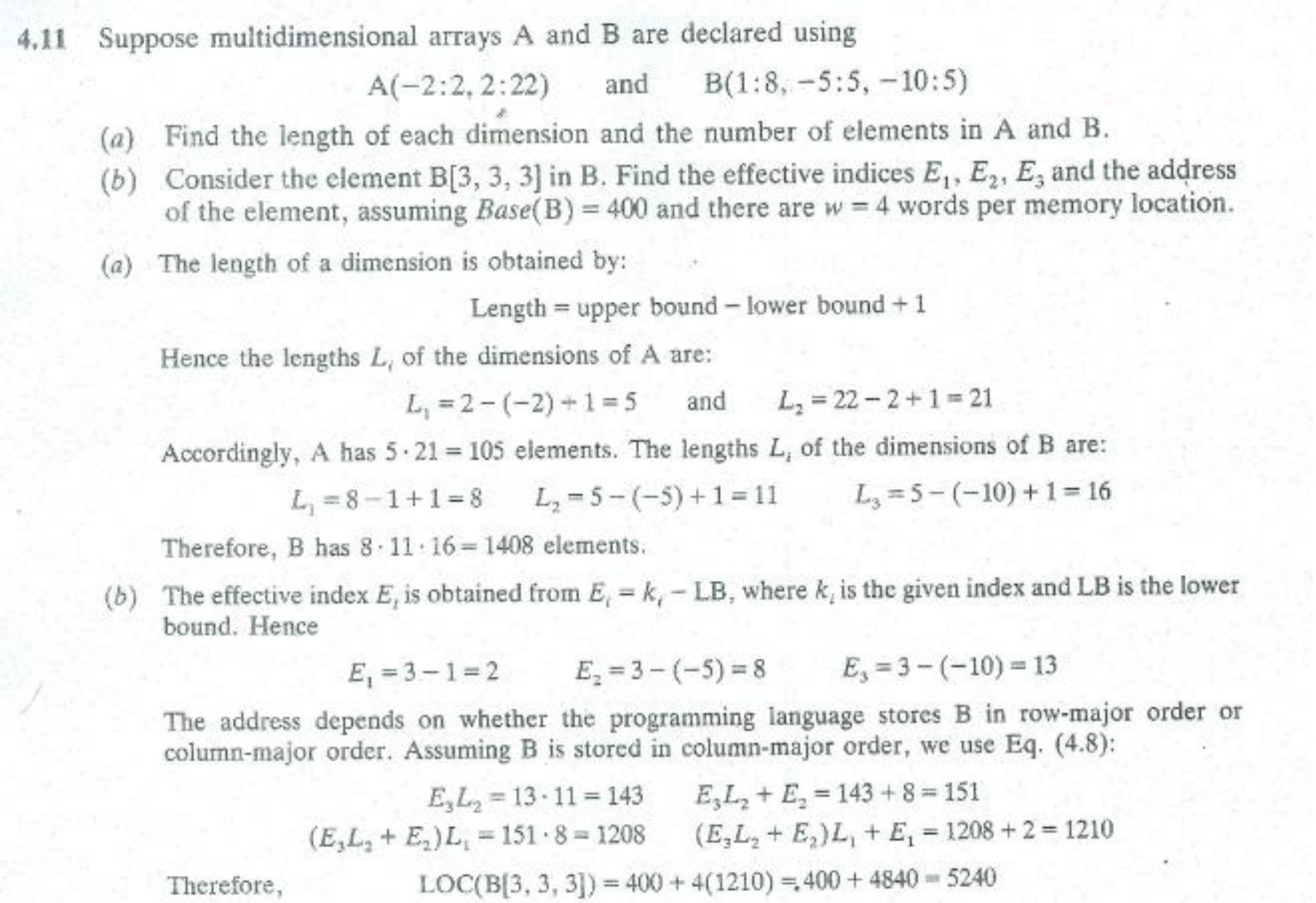


```
(a)  1. Set NUM := 0.
     2. Repeat for I = 1 to N:
     3.     Repeat for J = 1 to N:
                If A[I, J] ≠ 0, then: Set NUM := NUM + 1.
            [End of inner loop.]
        [End of outer loop.]
     4. Return.

(b)  1. Set SUM := 0.
     2. Repeat for J = 2 to N:
     3.     Repeat for I = 1 to J − 1:
                Set SUM := SUM + A[I, J].
            [End of inner Step 3 loop.]
     4. Return.

(c)  1. Set PROD := 1. [This is analogous to setting SUM = 0.]
     2. Repeat for K = 1 to N:
            Set PROD := PROD * A[K, K].
        [End of loop.]
     3. Return.
```

Example



4.11 Suppose multidimensional arrays A and B are declared using

$$A(-2:2, 2:22) \quad \text{and} \quad B(1:8, -5:5, -10:5)$$

(*a*) Find the length of each dimension and the number of elements in A and B.

(*b*) Consider the element B[3, 3, 3] in B. Find the effective indices E_1, E_2, E_3 and the address of the element, assuming *Base*(B) = 400 and there are $w = 4$ words per memory location.

(*a*) The length of a dimension is obtained by:

$$\text{Length} = \text{upper bound} - \text{lower bound} + 1$$

Hence the lengths L_i of the dimensions of A are:

$$L_1 = 2 - (-2) + 1 = 5 \quad \text{and} \quad L_2 = 22 - 2 + 1 = 21$$

Accordingly, A has $5 \cdot 21 = 105$ elements. The lengths L_i of the dimensions of B are:

$$L_1 = 8 - 1 + 1 = 8 \qquad L_2 = 5 - (-5) + 1 = 11 \qquad L_3 = 5 - (-10) + 1 = 16$$

Therefore, B has $8 \cdot 11 \cdot 16 = 1408$ elements.

(*b*) The effective index E_i is obtained from $E_i = k_i - \text{LB}$, where k_i is the given index and LB is the lower bound. Hence

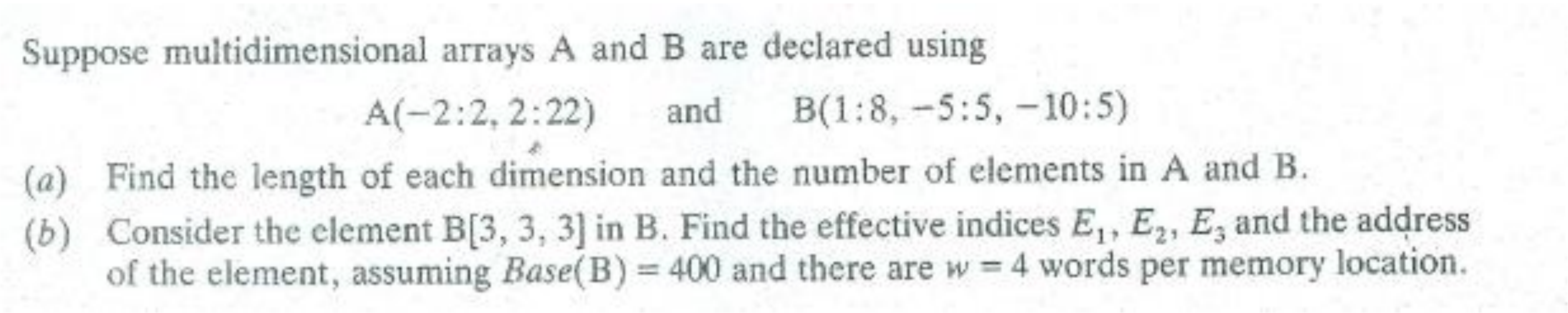
$$E_1 = 3 - 1 = 2 \qquad E_2 = 3 - (-5) = 8 \qquad E_3 = 3 - (-10) = 13$$

The address depends on whether the programming language stores B in row-major order or column-major order. Assuming B is stored in column-major order, we use Eq. (4.8):

$$E_3L_2 = 13 \cdot 11 = 143 \qquad E_3L_2 + E_2 = 143 + 8 = 151$$
$$(E_3L_2 + E_2)L_1 = 151 \cdot 8 = 1208 \qquad (E_3L_2 + E_2)L_1 + E_1 = 1208 + 2 = 1210$$

Therefore, $\text{LOC}(B[3, 3, 3]) = 400 + 4(1210) = 400 + 4840 = 5240$

Example



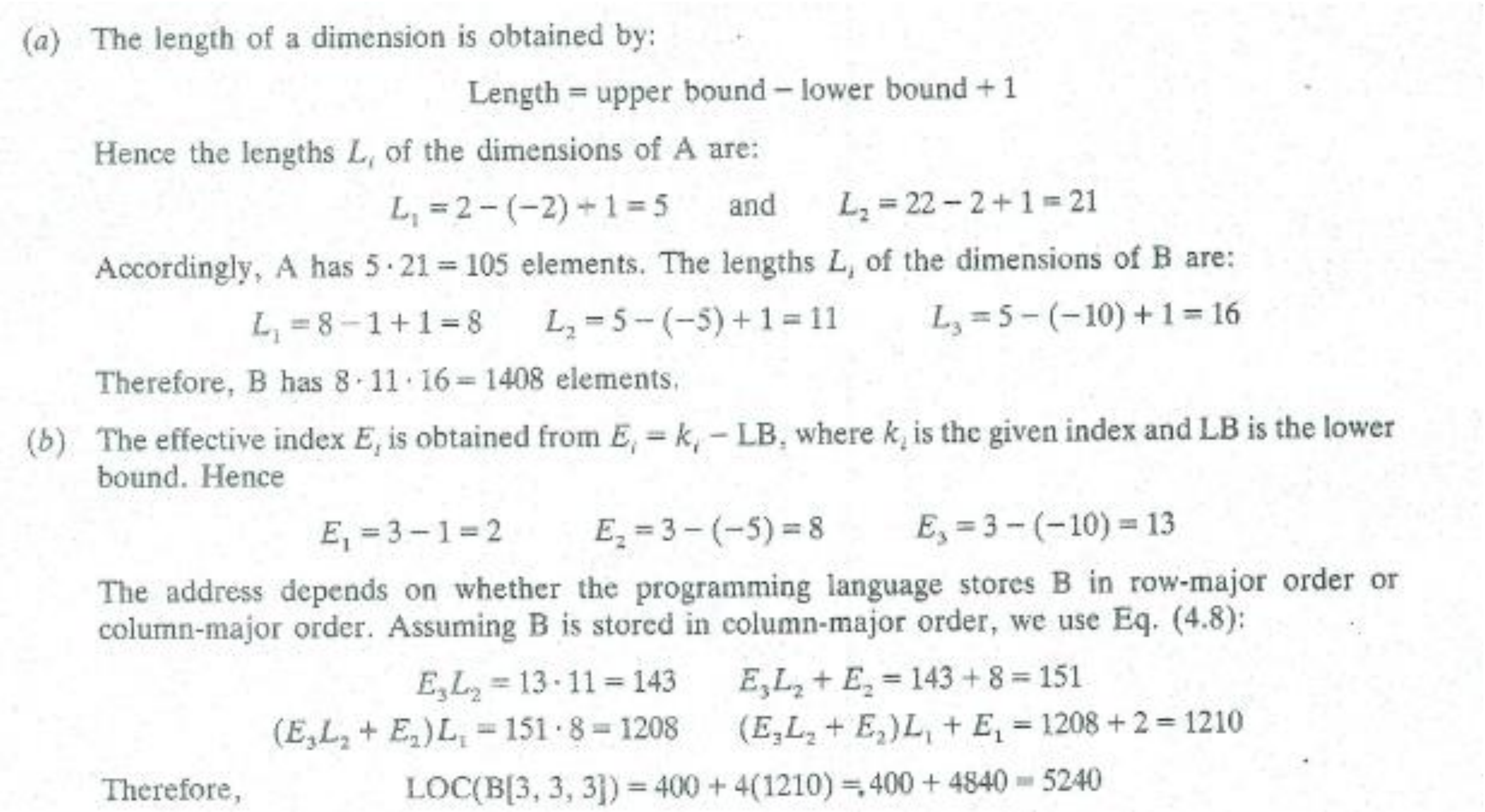
Suppose multidimensional arrays A and B are declared using

$$A(-2:2, 2:22) \qquad \text{and} \qquad B(1:8, -5:5, -10:5)$$

(*a*) Find the length of each dimension and the number of elements in A and B.

(*b*) Consider the element B[3, 3, 3] in B. Find the effective indices E_1, E_2, E_3 and the address of the element, assuming $Base(B) = 400$ and there are $w = 4$ words per memory location.

Example



(*a*) The length of a dimension is obtained by:

$$\text{Length} = \text{upper bound} - \text{lower bound} + 1$$

Hence the lengths L_i of the dimensions of A are:

$$L_1 = 2 - (-2) + 1 = 5 \qquad \text{and} \qquad L_2 = 22 - 2 + 1 = 21$$

Accordingly, A has $5 \cdot 21 = 105$ elements. The lengths L_i of the dimensions of B are:

$$L_1 = 8 - 1 + 1 = 8 \qquad L_2 = 5 - (-5) + 1 = 11 \qquad L_3 = 5 - (-10) + 1 = 16$$

Therefore, B has $8 \cdot 11 \cdot 16 = 1408$ elements.

(*b*) The effective index E_i is obtained from $E_i = k_i - \text{LB}$, where k_i is the given index and LB is the lower bound. Hence

$$E_1 = 3 - 1 = 2 \qquad E_2 = 3 - (-5) = 8 \qquad E_3 = 3 - (-10) = 13$$

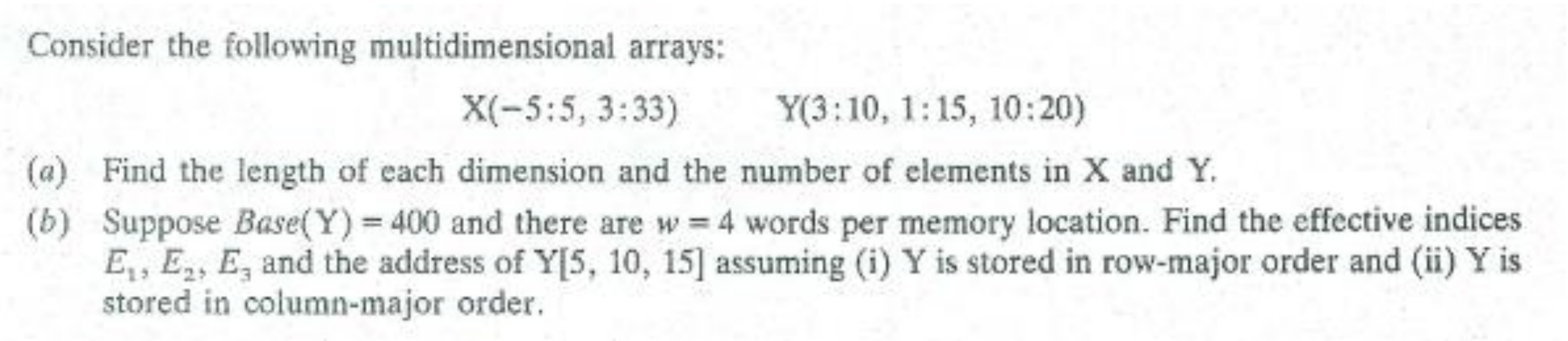
The address depends on whether the programming language stores B in row-major order or column-major order. Assuming B is stored in column-major order, we use Eq. (4.8):

$$E_3L_2 = 13 \cdot 11 = 143 \qquad E_3L_2 + E_2 = 143 + 8 = 151$$

$$(E_3L_2 + E_2)L_1 = 151 \cdot 8 = 1208 \qquad (E_3L_2 + E_2)L_1 + E_1 = 1208 + 2 = 1210$$

Therefore, $\quad \text{LOC}(\text{B}[3, 3, 3]) = 400 + 4(1210) = 400 + 4840 = 5240$

Problem



Consider the following multidimensional arrays:

$$X(-5:5,\ 3:33) \qquad Y(3:10,\ 1:15,\ 10:20)$$

(*a*) Find the length of each dimension and the number of elements in X and Y.

(*b*) Suppose *Base*(Y) = 400 and there are $w = 4$ words per memory location. Find the effective indices E_1, E_2, E_3 and the address of Y[5, 10, 15] assuming (i) Y is stored in row-major order and (ii) Y is stored in column-major order.

4.13. Matrices

Matrices

1. Algebra of Matrices
2. Example 4.23
3. Algorithm 4.7 (Matrix Multiplication)
4. Example 4.24

Matrices Multiplication Algorithm

- **Input two matrixes, Output Output matrix C.**
- **Matrix-Multiply(A, B)**

```
if columns [A] $\neq$ rows [B]
    then error "incompatible dimensions"
else
    for i =1 to rows [A]
        for j = 1 to columns [B]
            $C[i, j] = 0$
            for k = 1 to columns [A]
                $C[i, j] = C[i, j] + A[i, k] * B[k, j]$
return C
```

- **Complexity $O(n^3)$**

Algorithm Description

- To multiply two matrixes sufficient and necessary condition is "number of columns in matrix A = number of rows in matrix B".
- Loop for each row in matrix A.
- Loop for each columns in matrix B and initialize output matrix C to 0.
- This loop will run for each rows of matrix A.
- Loop for each columns in matrix A.
- Multiply A[i,k] to B[k,j] and add this value to C[i,j]
- Return output matrix C.

تم الإنتهاء من المحاضرة الرابعة