Programming for Engineers
Fortran: Subscripted Variables and Arrays

Abu Hasan Abdullah

January 7, 2009

Overview

1. Arrays are a convenient way to work with large quantities of data
2. With arrays we only need a single variable to control 100 numbers
3. Without arrays we need 100 conventional single-valued variables
4. Each array has *index* to locate and manipulate quantities stored in the array
5. Index is sometimes called *subscript*

Course Text:
Mayo W. E. and Cwiakala M. (1995): *Programming with Fortran 77*, ISBN 0-07-041155-7, McGraw-Hill

Overview

This session focuses on

1. Need for arrays
2. Declarations and 1-D arrays
3. Manipulation of arrays
4. 2-D and higher order arrays
5. I/O of arrays

Need for Arrays

- Engineers and scientists often work with large amounts of data
- For instance, a program that reads in ten numbers and prints them may have

```
READ *, X1, X2, X3, X4, X5, X6, X7, X8, X9, X10
PRINT *, X1, X2, X3, X4, X5, X6, X7, X8, X9, X10
```

- *If we wanted to perform this on 1000 numbers, there'd be a lot of work*

Need for Arrays

- There is a difference between
  - *single-valued variable*, and
  - *subscripted variable*

- So far we used *single-valued variable*s that take only a single value:

```
X1 = 1.23456
X2 = 9.87654
```

- To create a *subscripted variable* we must add the *subscript*, for example

```
X(1) = 1.23456
X(2) = 9.87654
```

  number within parentheses is the *index*

Need for Arrays

- The array *subscript* or *index* can be controlled by a DO loop using *LCV*

```
DO I = 1, 10
   X(I) = I**2
END DO
```

- Turn Example 6.1 into a complete program, compile and run it to study how DO loop populate an array

Array Declaration Statement

- General form of array declaration

  `type arrayname (lower limit : upper limit)`

  where

`type`	type of array (REAL, INTEGER, LOGICAL, etc.)
`arrayname`	valid Fortran variable name
`lower limit`	lowest value for the subscript
`upper limit`	maximum value for the subscript

- `lower limit:upper limit` of numbers can be any integers (even negative)

- `upper limit` MUST be larger than `lower limit`

Array Declaration Statement

- Examples

```
REAL X(1:10)
REAL Y(-5:20)
INTEGER COUNTER(20)
```

Manipulating Arrays

- 1-D arrays are efficient way to store and manipulate list and tables of data
- Control by loops are ideal for storage and manipulation of arrays
- Loops and arrays are therefore almost inseparable

Manipulating Arrays

- For example, take the sum of individual elements

$$a = \sum_{i=1}^{i=100} y_i$$

- . . . can be implemented in Fortran with a DO loop thus

```
A = 0.0
DO I = 1, 100
   A = A + Y(I)
END DO
```

The DO loop takes each element of the array and adds it to the running total

Manipulating Arrays

- Turn Examples 6.4–6.10 into complete programs, compile and run them to study various array manipulation using `DO` loops

Application - Summation of Array Elements

EXAMPLE 6.4

Sigma notation (Σ) is used very frequently in engineering, science, and mathematics as a shorthand notation. So we will explore this subject in considerable detail in this and subsequent examples. Recall that sigma notation indicates the sum of the individual elements of the subscripted expression is to be computed. Here is a simple example:

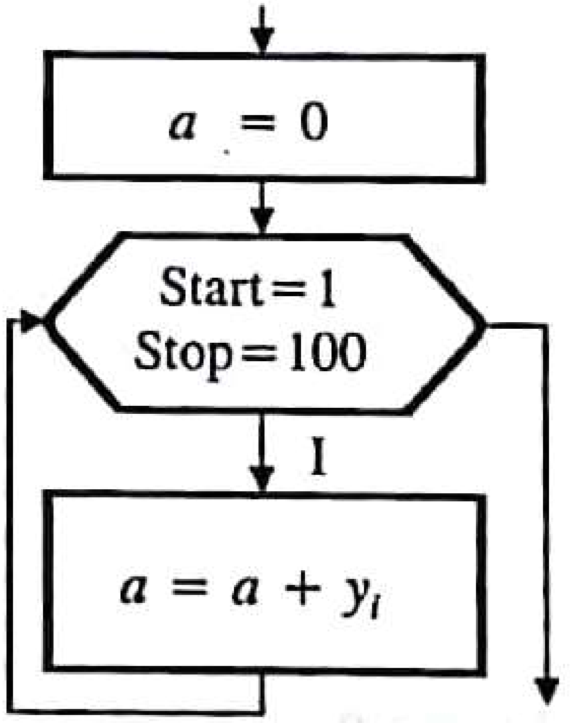
$$a = \sum_{i=1}^{i=100} y_i$$

This expression represents the sum of all the elements stored in the subscripted variable y and the statement stores the result in a. We add one element of the array at a time, and systematically change the subscript from $i=1$ to $i=100$. We implement this in Fortran with a DO loop (we leave out the input statements since we haven't discussed I/O of arrays yet):

Program

```
      REAL Y(100)
C We have left out the input
C statements that assign
C values to Y
      A = 0.0
      DO 10 I = 1, 100
         A = A + Y(I)
10    CONTINUE
```

Flowchart



EXAMPLE 6.5

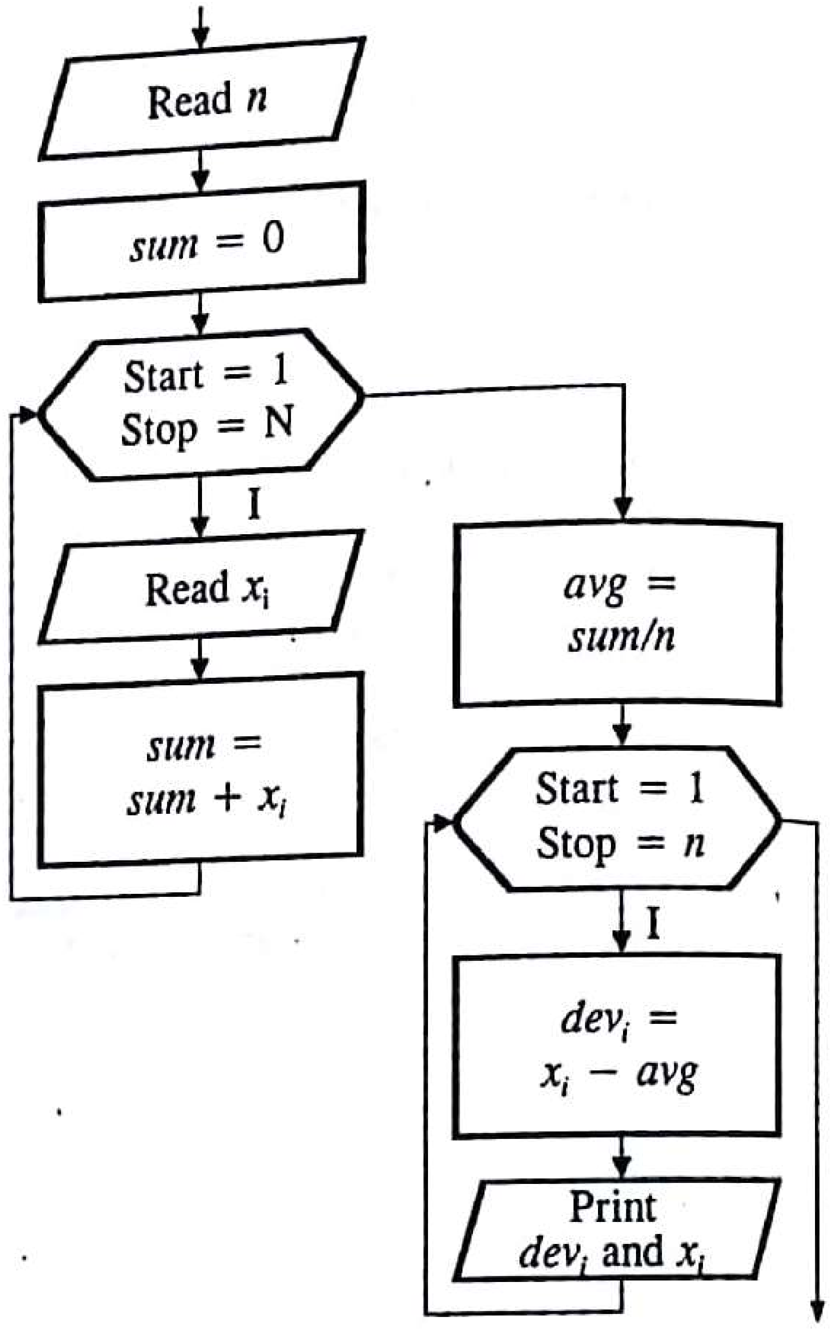
Create a program that reads in a list of numbers from the terminal, calculates the average, and finally prints a list of the individual deviations of each number from the average. The deviation is the difference between the number and the average. Assume a maximum of 100 numbers will be entered.

To solve this problem, we will use two arrays: X and DEV. The X array will store the numbers as we enter them. Once all the numbers are entered, we will be able to compute the average. Finally, we can then compute the deviations by subtracting the average value from each of the input numbers. Note that we have to save the entered numbers so that we can use them a second time for the computation of the deviations. A simple algorithm and flowchart to do this are:

Algorithm

```
1. Read in number of data points, N
2. SUM = 0.0
3. Loop (1 to N)
       Read in a value and assign to X(I)
       Add X(I) to SUM
4. Compute average (AVG = SUM/N)
5. Loop (1 to N)
       DEV(I) = X(I) - AVG
       Print X(I) and DEV(I)
```

Flowchart



Program

```
      REAL X(100), DEV(100)
C Enter the number of data items for the computation
      PRINT *, 'Number of values (less than 100)?'
      READ *, N
C We will read in one data value at a time and store it in X(I)
      SUM = 0.0
      DO 10 I = 1, N
           READ *, X(I)
           SUM = SUM + X(I)
 10   CONTINUE
      AVG = SUM/N
C Once the average has been computed, we can use it to
C calculate the deviations defined by X(I)-AVG:
      PRINT *, 'Average = ', AVG
      DO 20 I = 1, N
           DEV(I) = X(I) - AVG
           PRINT *, 'NUMBER=', X(I)
           PRINT *, 'DEVIATION=', DEV(I)
 20   CONTINUE
      END
```

Application – Dot Product of Vectors

EXAMPLE 6.6

The *dot product* z of two vectors a and b is defined by:

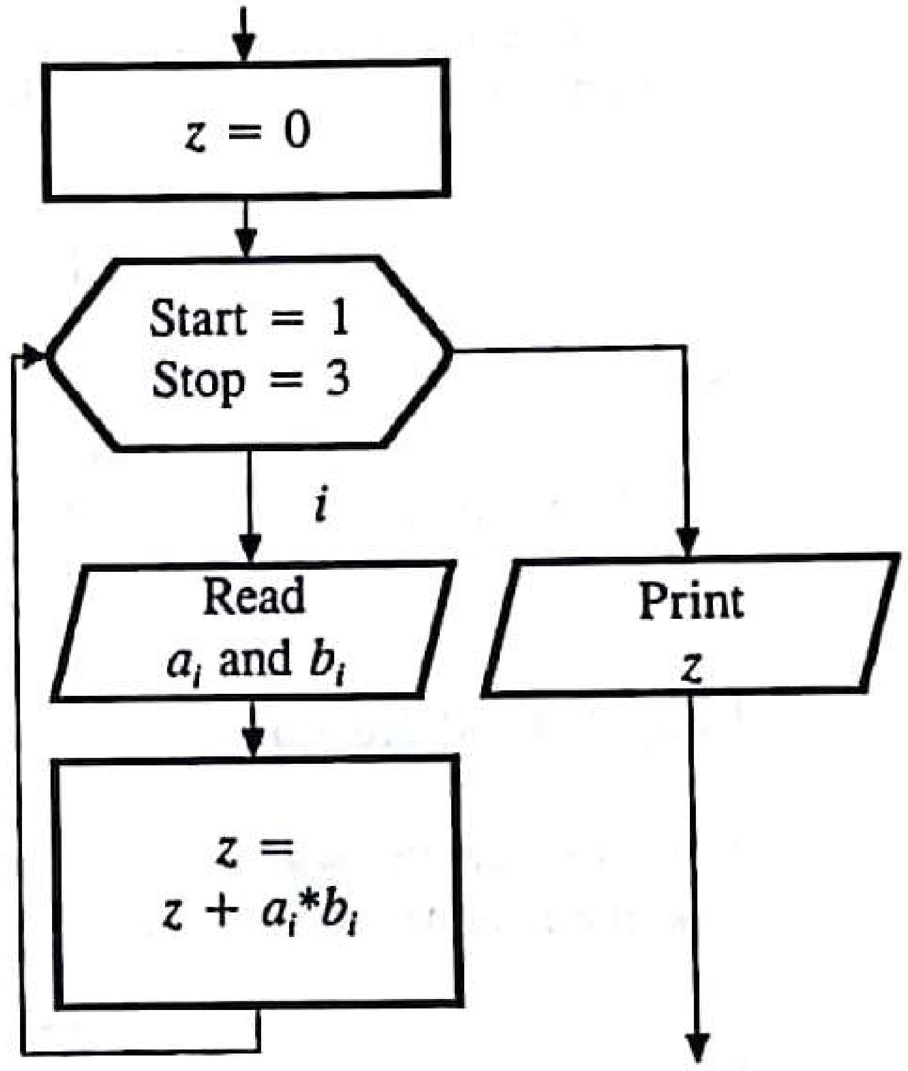
$$z = \vec{a} \odot \vec{b} = \sum_{i=1}^{i=3} a_i b_i$$

We start by noting that the summation process can be implemented with a single DO loop as shown in the previous example. Inside the loop, we will add the product of the appropriate components of each vector. For example, for the two vectors $a = (1.2, 3.5, 4.1)$ and $b = (2.0, 5.1, -1.1)$, the dot product is given by $1.2 \times 2.0 + 3.5 \times 5.1 + 4.1 \times (-1.1) = 2.4 + 17.85 - 4.51 = 15.74$. The algorithm for doing this is:

Algorithm

```
1. Z = 0.0
2. Loop (1 to 3)
       Read $A_I$ and $B_I$
       Add $A_I \times B_I$ to Z
3. Print Z
```

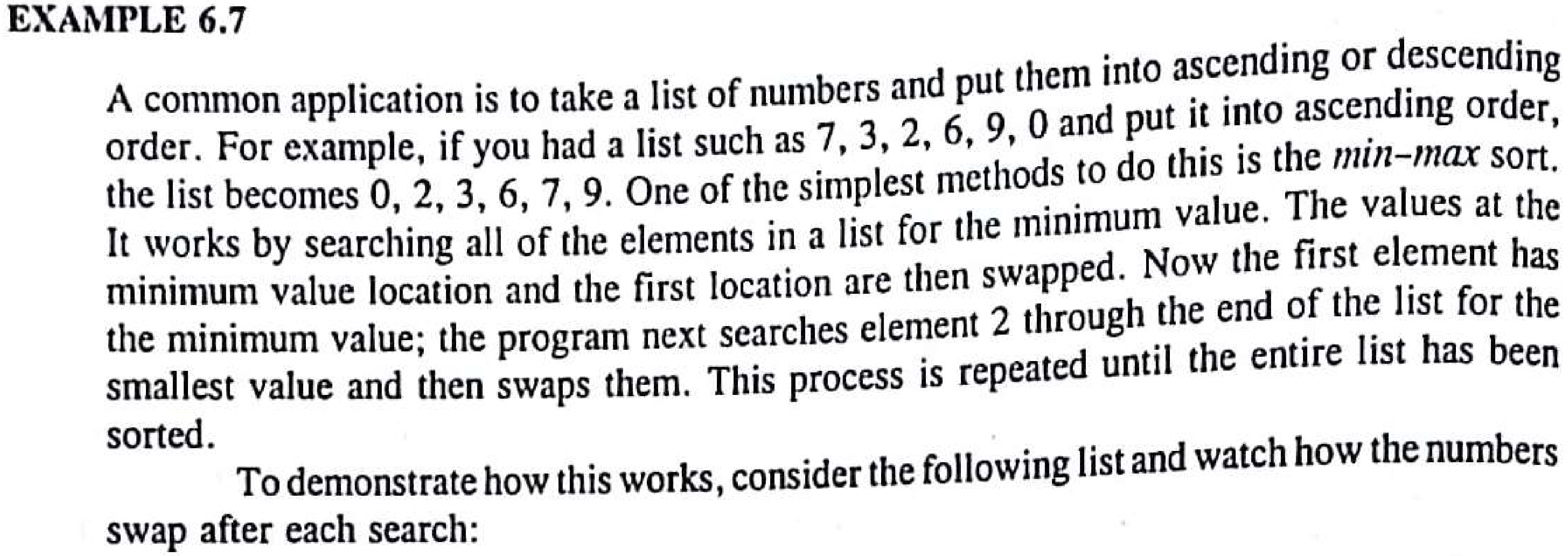
Flowchart



Program

```
C The two vectors A and B each contain 3 components. So
C we declare each to be a one-dimensional array with 3 elements.
      REAL A(3), B(3)
      Z = 0.0
C Inside the following loop, we read in the components of each
C vector and perform the required summation of the products.
      DO 10 I = 1, 3
           READ *, A(I), B(I)
           Z = Z + A(I) * B(I)
 10   CONTINUE
      PRINT*, 'Dot Product = ', Z
      END
```

Application – Sorting of Lists



EXAMPLE 6.7

A common application is to take a list of numbers and put them into ascending or descending order. For example, if you had a list such as 7, 3, 2, 6, 9, 0 and put it into ascending order, the list becomes 0, 2, 3, 6, 7, 9. One of the simplest methods to do this is the *min–max* sort. It works by searching all of the elements in a list for the minimum value. The values at the minimum value location and the first location are then swapped. Now the first element has the minimum value; the program next searches element 2 through the end of the list for the smallest value and then swaps them. This process is repeated until the entire list has been sorted.

To demonstrate how this works, consider the following list and watch how the numbers swap after each search:

Starting Values	7	3	2	6	9	0

We assume that the minimum value is in the first position. But, as we search through the list, we find the smallest value is in the sixth position. So, we switch the first and sixth values:

Swap 1st and 6th values	0	3	2	6	9	7

Now we start the search at the second position (since we know that the first position has the smallest value). We assume that the minimum value is in the second position. But, we find that the value in the third position is smallest (in the shortened list), so we swap the values in the second and third positions:

Swap 2nd and 3rd values	0	2	3	6	9	7

Now we start the search at the third position, assuming that its value is the smallest. This time the assumption is correct, so we do nothing:

Leave 3rd value alone	0	2	3	6	9	7

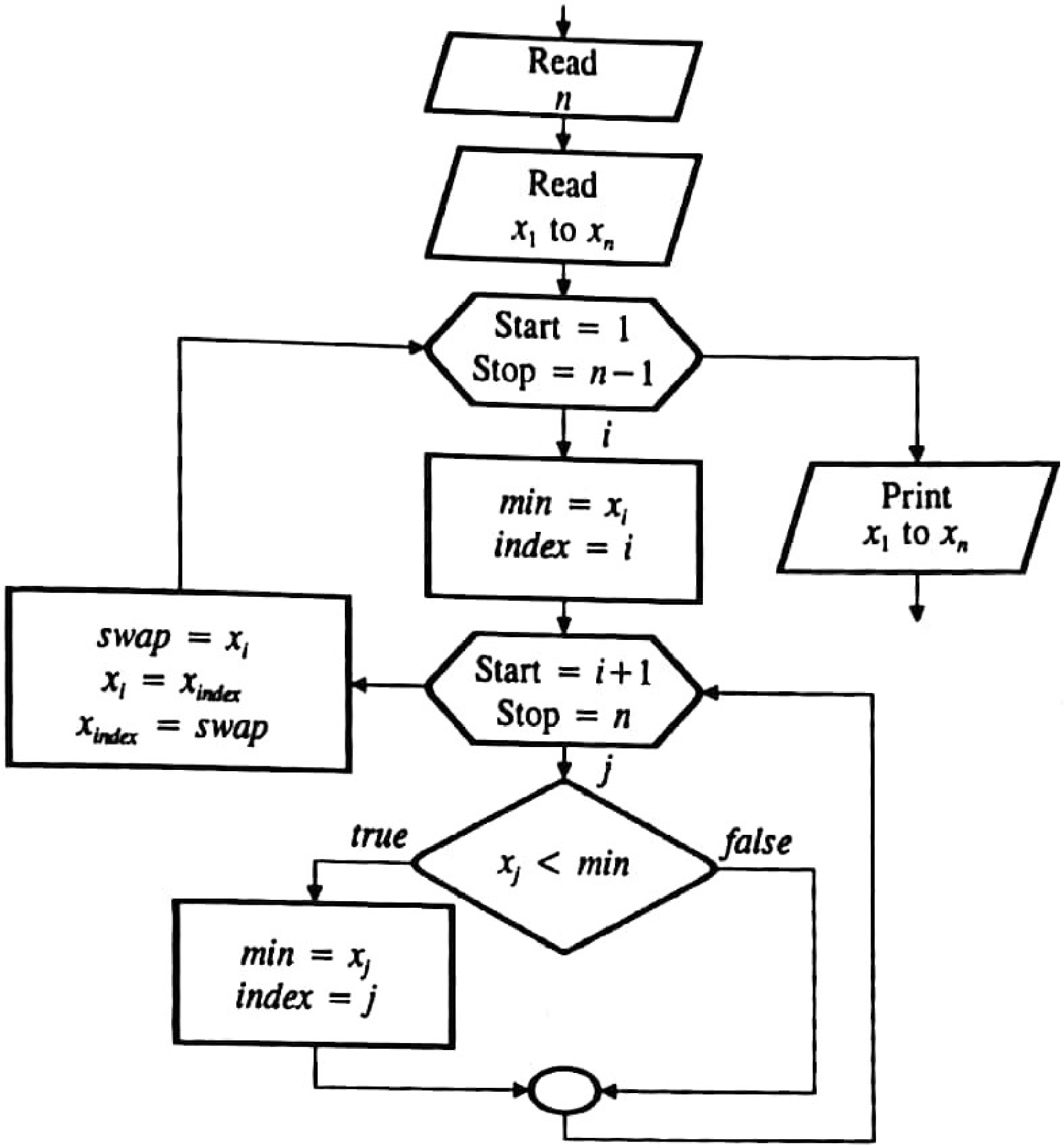
Now we start the search at the fourth position, assuming that its value is the smallest. Once again, the assumption is correct, so we do nothing:

Leave 4th value alone	0	2	3	6	9	7

Next, we start at the fifth position, and we find that we must swap the fifth and sixth values:

Swap 5th and 6th values	0	2	3	6	7	9

Flowchart



Read n
Read x_1 to x_n
Start = 1
Stop = $n-1$
i
$min = x_i$
$index = i$
Print x_1 to x_n
$swap = x_i$
$x_i = x_{index}$
$x_{index} = swap$
Start = $i+1$
Stop = n
j
$x_j < min$
true
false
$min = x_j$
$index = j$

Program

```
      REAL X(100)
      PRINT *, 'How many numbers?'
C Input N values into X(1) to X(N)
      READ *, N
      DO 10 I = 1, N
        READ *, X(I)
 10   CONTINUE
C The DO 30 loop continues until all the values are sorted
      DO 30 I = 1, N-1
        XMIN = X(I)
        INDEX = I
C The Do 20 loop locates the position of the smallest value
        DO 20 J = I+1, N
          IF (X(J) .LT. XMIN) THEN
            XMIN = X(J)
            INDEX = J
          END IF
 20     CONTINUE
C The following section swaps the value in X(I) and the
C smallest value located in X(INDEX)
        SWAP = X(I)
        X(I) = X(INDEX)
        X(INDEX) = SWAP
 30   CONTINUE
      DO 40 I = 1, N
        PRINT *, X(I)
 40   CONTINUE
```

Application – Count of Certain Values

EXAMPLE 6.8

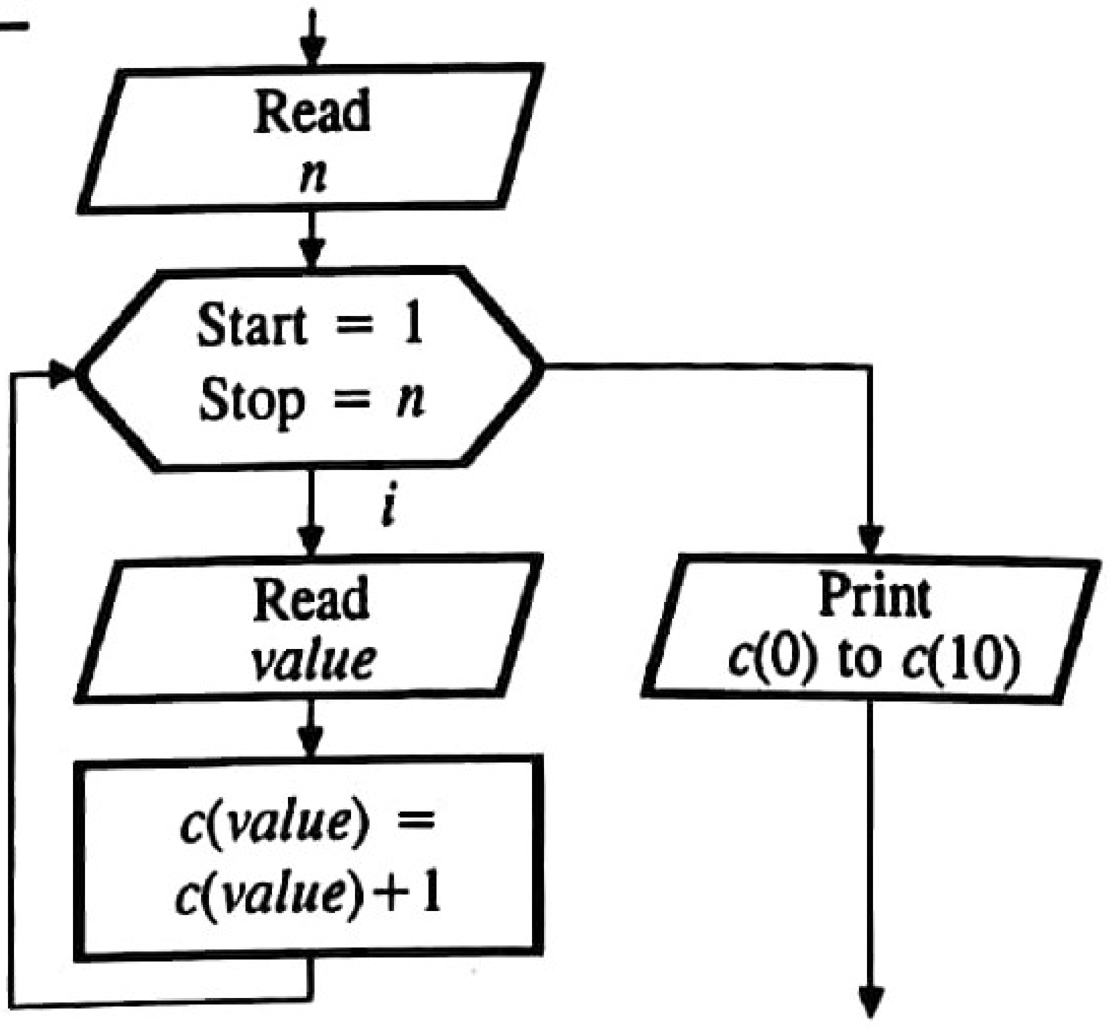
A data set contains a list of integers ranging from 0 to 10 where the same number may be entered multiple times. The following program counts the number of occurrences for each value. We will create an array C(I) that contains a count of each number as it is entered. Thus, C(0) represents the number of zeros, C(1) represents the number of ones, and so forth:

C(0)	C(1)	C(2)	C(3)	C(4)	C(5)	C(6)	C(7)	C(8)	C(9)	C(10)

When we read in a value, we will put that number into the appropriate bin. For example, if we read in the number 5, we would increase the count in C(5) by one:

C(0)	C(1)	C(2)	C(3)	C(4)	C(5)	C(6)	C(7)	C(8)	C(9)	C(10)
					1					

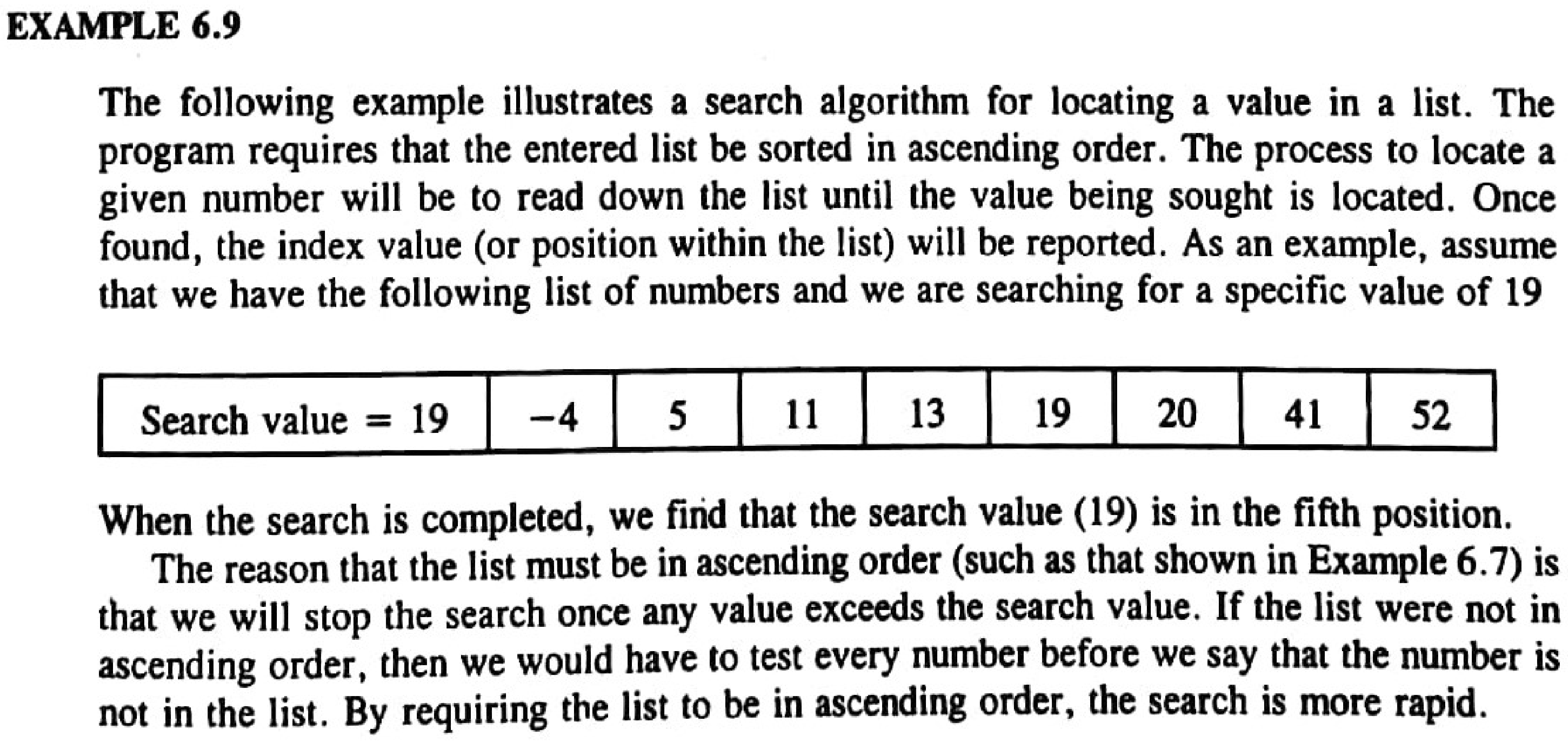
Flowchart


Read
n
Start = 1
Stop = n
i
Read
value
$c(value) = c(value)+1$
Print
$c(0)$ to $c(10)$

Program

```
C When we declare the array C, it makes sense to declare it to
C start at C(0). Note also that VALUE must be declared as an
C integer since we will be using it later as a subscript to store
C a value in C(VALUE)
      INTEGER C(0:10), VALUE
      PRINT *, 'Number of values?'
      READ *, N
C First, we initialize all elements of C to zero.
      DO 10 I = 0, 10
        C(I) = 0
 10   CONTINUE
C Read in a value and increment the appropriate list position.
      DO 20 I = 1, N
        READ *, VALUE
        C(VALUE) = C(VALUE) + 1
 20   CONTINUE
C Print out the results
      DO 40 I = 0, 10
        PRINT *, 'Number of', I, '''s were', C(I)
 40   CONTINUE
      END
```

Application – Locating/Location of Certain Values in a List



EXAMPLE 6.9

The following example illustrates a search algorithm for locating a value in a list. The program requires that the entered list be sorted in ascending order. The process to locate a given number will be to read down the list until the value being sought is located. Once found, the index value (or position within the list) will be reported. As an example, assume that we have the following list of numbers and we are searching for a specific value of 19

Search value = 19	−4	5	11	13	19	20	41	52

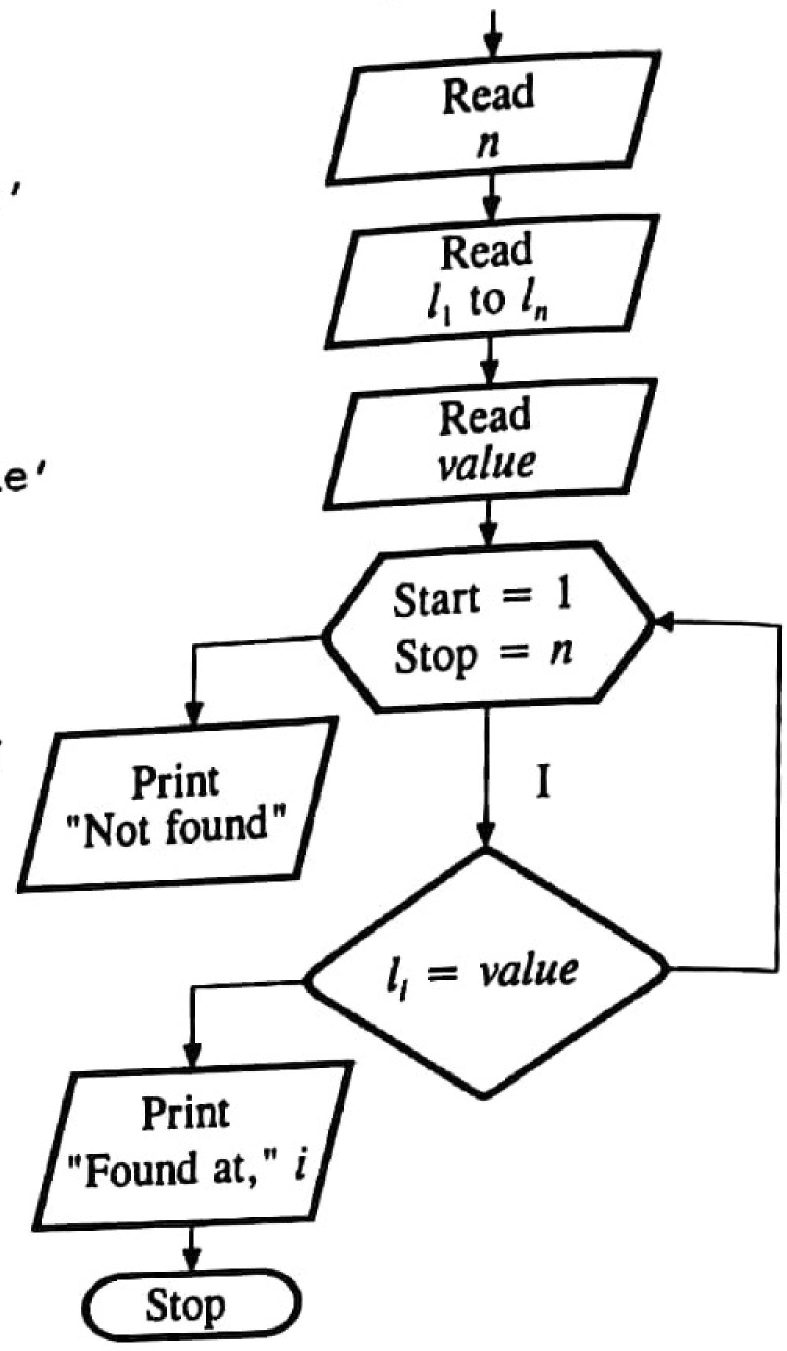
When the search is completed, we find that the search value (19) is in the fifth position.

The reason that the list must be in ascending order (such as that shown in Example 6.7) is that we will stop the search once any value exceeds the search value. If the list were not in ascending order, then we would have to test every number before we say that the number is not in the list. By requiring the list to be in ascending order, the search is more rapid.

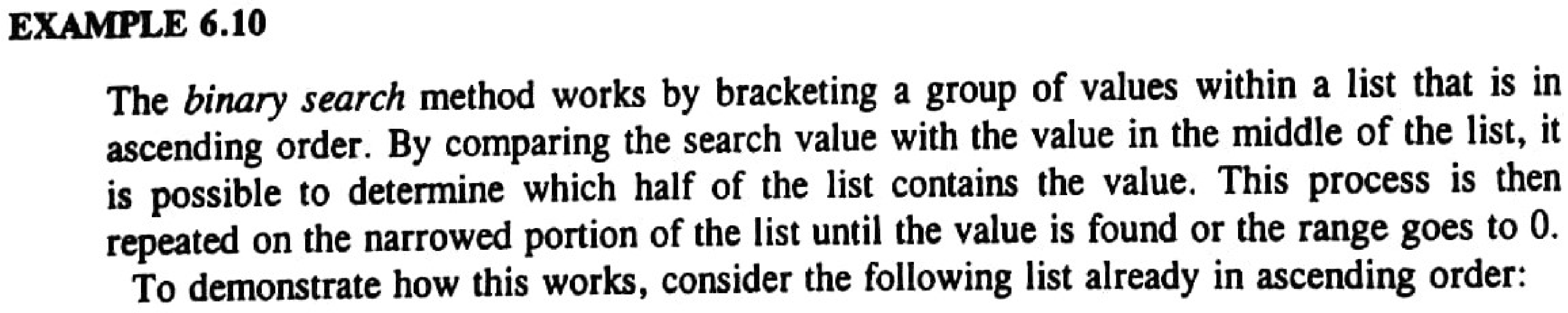
Program

```
C Declare the list as L(100)
        INTEGER L(100), VALUE
        PRINT *, 'Number of values?'
        READ *, N
        DO 10 I = 1, N
          READ *, L(I)
 10     CONTINUE
C Enter the value to be sought
        PRINT *, 'Enter search value'
        READ *, VALUE
C Check every value of L(I). If
C any value equals VALUE, then
C print out location and stop.
        DO 40 I = 1, N
          IF (L(I) .EQ. VALUE) THEN
            PRINT *, 'Found at', I
            STOP
          END IF
 40     CONTINUE
        PRINT *, 'Not found'
        END
```

Flowchart



Application – Binary Search for Locating/Location of Certain Values in a List

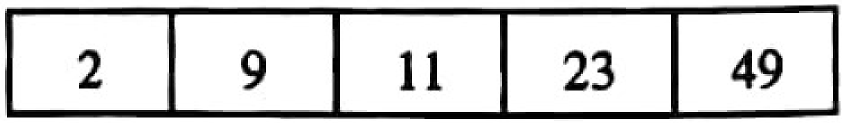


EXAMPLE 6.10

The *binary search* method works by bracketing a group of values within a list that is in ascending order. By comparing the search value with the value in the middle of the list, it is possible to determine which half of the list contains the value. This process is then repeated on the narrowed portion of the list until the value is found or the range goes to 0.

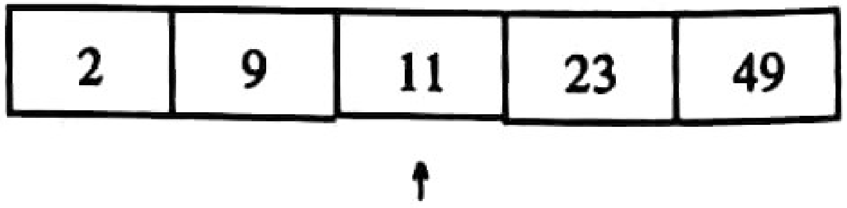
To demonstrate how this works, consider the following list already in ascending order:

Starting values



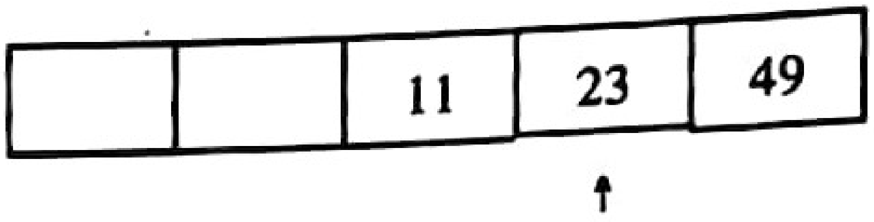
We start by comparing the search value (*e.g.*, 23), with the value at the center of the list:

Compare search value (23) with midpoint value



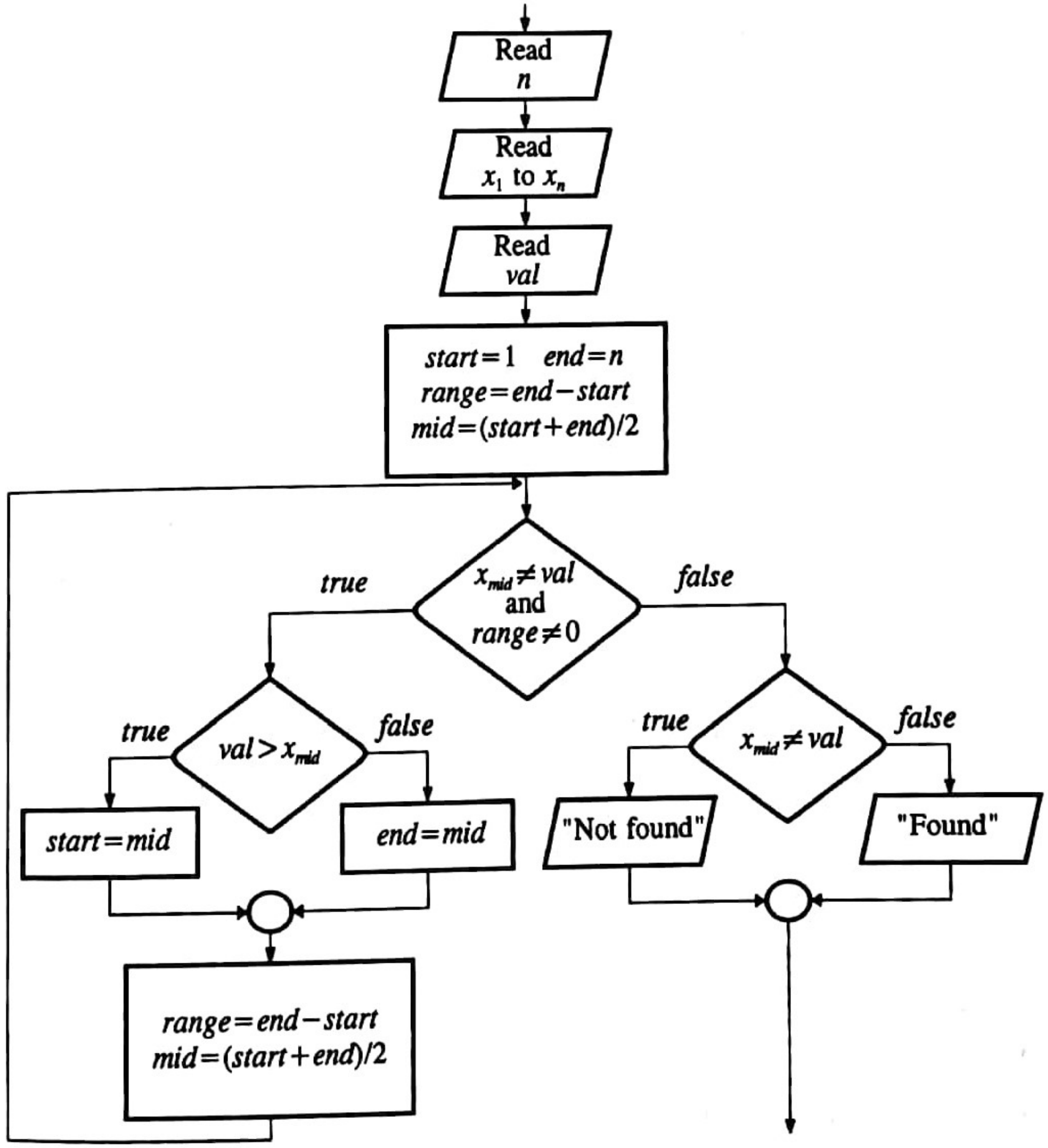
If the search value (23) is less than the value in this position, then the number is in the lower half of the list; otherwise, it is in the upper half. In this case, the search value (23) is larger than the value in the middle (11), so the search will focus on the second half. We then repeat the process by dividing the remaining numbers into two halves:

Cut search area in half and repeat previous step



The value in the center of the reduced search area is now equal to the search value (23), so our search stops. The program will then print out that it found the value in the fourth position. If the search area ever goes to zero, then the search number is not in the list.

Flowchart



Read n
Read x_1 to x_n
Read val
$start=1 \quad end=n$
$range=end-start$
$mid=(start+end)/2$
$x_{mid} \neq val$ and $range \neq 0$
true
false
$val > x_{mid}$
true
false
$start=mid$
$end=mid$
$range=end-start$
$mid=(start+end)/2$
$x_{mid} \neq val$
true
false
"Not found"
"Found"

Program

```
      INTEGER X(100)
      INTEGER RANGE
      INTEGER START, END
C The input section
      PRINT *, 'Number of values?'
      READ *, N
      DO 10 I = 1, N
        READ *, X(I)
 10   CONTINUE
C Enter the value to be sought
      PRINT *, 'Enter value'
      READ *, VAL
C Define the range and midpoint
      START = 1
      END = N
      RANGE = END - START
      MID = (START + END)/2
C As long as the value is not found, cut the range in half
C and check which half the value might be in.
      DO WHILE (X(MID) .NE. VAL .AND. RANGE .NE. 0)
         IF (VAL .GT. X(MID)) THEN
           START = MID
         ELSE
           END = MID
         ENDIF
         RANGE = END - START
         MID = (START + END)/2
      END DO
C If the value being sought is not in the middle of the last
C range, then the value is not in the original list.
      IF (X(MID) .NE. VAL) THEN
           PRINT *, VAL, 'not found'
      ELSE
           PRINT *, 'Value at', MID
      ENDIF
      END
```

2-D and Higher-order Arrays

- Array dimension could be thought of as the number of subscripts required to locate a value stored in the array
- 1-D array represents a *list of variables*
- 2-D array can be thought of as representing a *table of information*
- To select a value from a table, we need to specify the row and column. Thus, two subscripts are required
- Processing of 2-D arrays is similar to that of 1-D arrays except that the is a second index

A
4
7
18

B		
3	-2	30
47	15	0
70	-8	39

Since a list requires only one subscript to locate a value, we can see immediately that A(2) has the value 7. A table, however, requires two subscripts to locate a value. The only question is which subscript (row or column) comes first? Fortran adopts the convention that the row will come first. Thus, B(2,3) has the value 0. Similarly, B(3,2) has the value –8. Note very carefully that B(3,2) is not equal to B(2,3). If we wanted to add the values in the second column, we could do this with B(1,2) + B(2,2) + B(3,2) = 5. As we did when processing one-dimensional arrays, we will use subscripts to process the data contained within an array. But this time, we will need two subscripts.

2-D and Higher-order Arrays

- Turn Examples 6.11–6.14 into complete programs, compile and run them to study various 2-D array manipulation

Application – Matrix Multiplication

EXAMPLE 6.14

The process of multiplying two arrays together can be expressed using sigma notation.

$$c_{ij} = \sum_k a_{ik} b_{kj}$$

This expression indicates how to calculate each entry in the product array c. Note that the number of columns in the a array must match the number of rows in the b array. The resulting array will have the same number of rows as a and the same number of columns as b. To illustrate how this summation works, let's compute the term c_{12}. This will require the summation of the individual products of $a_{1k}b_{k2}$. Assume for example, that we have the a matrix of size 3×2 and the b matrix of size 2×3, which when multiplied produce a c matrix with 3 columns and 3 rows:

$$a = \begin{bmatrix} 1 & 2 \\ 4 & 6 \\ 1 & 0 \end{bmatrix} \qquad b = \begin{bmatrix} 2 & 1 & 4 \\ 3 & 0 & 7 \end{bmatrix}$$

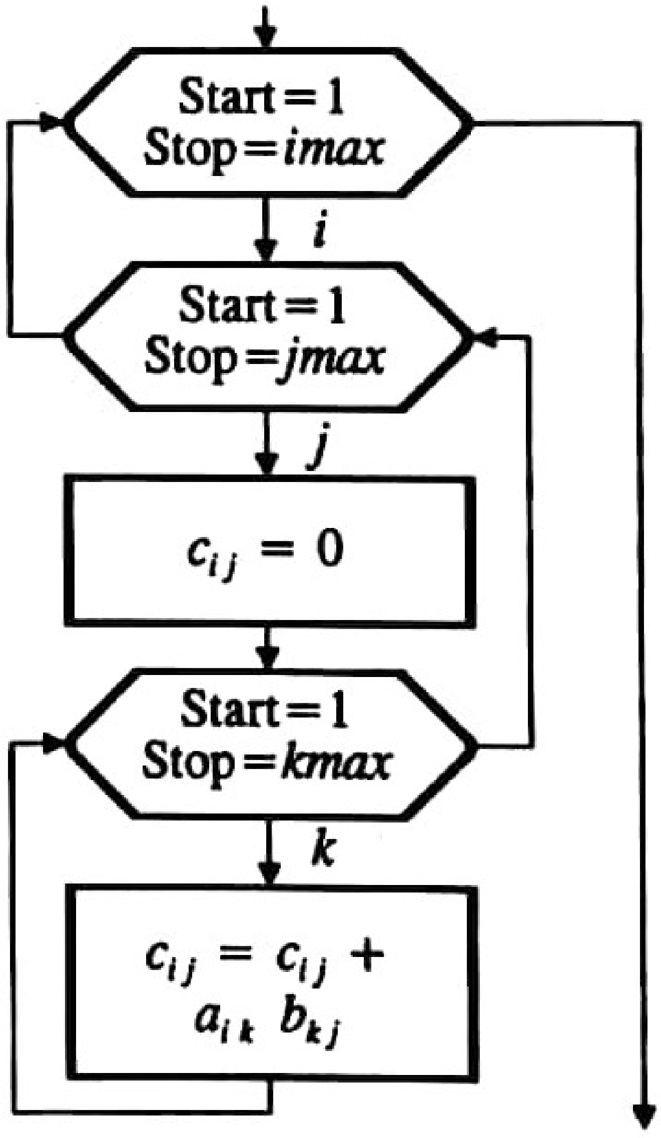
The desired term c_{12} would be $a_{11}b_{12} + a_{12}b_{22}$ or $(1)(1) + (2)(0) = 1$. In a similar way, we can generate all other elements in c, which will have three rows and three columns:

$$c = \begin{bmatrix} (1)(2) + (2)(3) & (1)(1) + (2)(0) & (1)(4) + (2)(7) \\ (4)(2) + (6)(3) & (4)(1) + (6)(0) & (4)(4) + (6)(7) \\ (1)(2) + (0)(3) & (1)(1) + (0)(0) & (1)(4) + (0)(7) \end{bmatrix}$$

Program

```
C   VARIABLE LISTING:
C     A,B   = Input Matrices
C     C     = Output Matrix
C     IMAX  = Number of rows in A
C     JMAX  = Number of Columns in B
C     KMAX  = Number of rows in B, and
C           = Number of columns in A
C-----------------------------------
              :
(assume A,B, and C already declared)
              :
        DO 10 I = 1, IMAX
          DO 20 J = 1, JMAX
            C(I,J) = 0.0
            DO 30 K = 1, KMAX
              C(I,J)=C(I,J)+A(I,K)*B(K,J)
30          CONTINUE
20        CONTINUE
10      CONTINUE
        END
```

Flowchart



I/O of Arrays

- The simplest is to refer to each element individually as one would with a single-valued variable (also called scalar)

```
PRINT *, X(1), X(2), Y(5)
```

  but this impractical for a long list of array elements

- We can instead use loop with I/O statement to simplify the process

```
      DO 10 I=1,100
         PRINT *, A(I)
10    CONTINUE
```

  with each encounter of a PRINT statement, a new line is created

I/O of Arrays

- There is also a special form of DO loop known as *implied DO loop*, limited to I/O and DATA statements

- General form of implied DO loop

```
I/O Command (array(LCV),LCV=start,stop[,step])
```

where

I/O Command	READ, PRINT or WRITE
LCV	loop control variable

- Sample code

```
READ *,(A(I), I = 1,10,1)
```

EXAMPLE 6.15

Listed below are examples of explicit forms of I/O and their equivalent implied DO loops:

Explicit Form	Implied Form	Comments
`      DO 10 I=1, 10`	`READ *,(A(I),I=1, 10)`	Reads 10 values into
`         READ *, A(I)`		array A.
`10    CONTINUE`		
`      DO 20 I=1, 10`	`READ *,(X(I),Y(I),I=1, 10)`	Reads in 10 sets of data
`         READ *, X(I), Y(I)`		as (x_1, y_1), (x_2, y_2),
`20    CONTINUE`		through (x_{10}, y_{10}).

Explicit Form	Implied Form	Comments
`      DO 30 I=1,20`	`READ *,((Z(I,J),J=1,10),I=1,20)`	An implied DO
`        DO 30 J=1,10`		loop within another
`          READ*, Z(I,J)`		implied DO loop is
`30    CONTINUE`		permitted.

I/O of Arrays

- Turn Examples 6.15–6.17 into complete programs, compile and run them to study various I/O operations on 2-D arrays

EXAMPLE 6.16

For the following example, A is a one-dimensional integer array of 10 elements and B is a 3 × 3 two-dimensional integer array. The value assigned to each element is:

A	1	2	3	4	5	6	7	8	9	10

B	1	2	3
	4	5	6
	7	8	9

	Program Segment	Output
a)	PRINT *, A(1), A(2), A(3)	1 2 3
b)	PRINT *, (A(I), I = 1, 3)	1 2 3
c)	PRINT *, (A(I), I = 2, 8, 2)	2 4 6 8
d)	PRINT *, ((B(I,J), I=1,3), J=1,3)	1 4 7 2 5 8 3 6 9
e)	PRINT *,B	1 4 7 2 5 8 3 6 9
f)	PRINT *, ((B(I,J), J=1,3), I=1,3)	1 2 3 4 5 6 7 8 9
g)	DO 10 I=1, 3	1 2 3
	PRINT *, (B(I,J), J=1,3)	4 5 6
10	CONTINUE	7 8 9
h)	PRINT *, ('+', I=1,10)	++++++++++

EXAMPLE 6.17

The following program will print out a crude graph of a sine wave over a range of 0 to 2π.

```
        DO 10 I = 0, 20
              X = I*2*3.1416/20.
              Y = SIN(X)
              N = NINT(30+Y*30+1)
              PRINT *, ('*', J = 1, N)
 10     CONTINUE
        END
```

Formatting of Array Output

EXAMPLE 6.18

In the following example, A is a one-dimensional real array of 10 elements and B is a 3 × 3 two dimensional real array. The value assigned to each element is:

A	1	2	3	4	5	6	7	8	9	10

B	1	2	3
	4	5	6
	7	8	9

```
Program Segment                                        Output

a)        PRINT 10, (A(I), I = 1, 5)
    10    FORMAT (' ', 5(F4.1, 1X))                     1.0 2.0 3.0 4.0 5.0

b)        PRINT 10, (A(I), I = 1, 5)
    10    FORMAT (' ', 20(F4.1, 1X))                    1.0 2.0 3.0 4.0 5.0

c)        PRINT 10, (A(I), I = 1, 10)                   1.0 2.0 3.0 4.0 5.0
    10    FORMAT (' ', 5(F4.1, 1X))                     6.0 7.0 8.0 9.0 10.0

d)        PRINT 10, ((B(I,J), J=1, 3), I=1, 3)         B:
    10    FORMAT (' B:', 3(/, 1X, 3(F4.1, 1X)))          1.0 2.0 3.0
                                                         4.0 5.0 6.0
                                                         7.0 8.0 9.0

e)        PRINT 10, ((B(I,J), J=1, 3), I=1, 3)          1.0 2.0 3.0
    10    FORMAT (' ', 3(F4.1, 1X))                     4.0 5.0 6.0
                                                        7.0 8.0 9.0

f)        N = 3                                          1.0 2.0 3.0
          DO 20 I = 1, N                                 4.0 5.0 6.0
             PRINT 10, (B(I,J), J = 1, N)                7.0 8.0 9.0
    10       FORMAT (' ', 100(F4.1, 1X))
    20    CONTINUE
```

PARAMETER and DATA Statements

- PARAMETER statement is an easy way of creating *named constants* . . .
- . . . whose value cannot be changed under any circumstances
- General form of array declaration

```
        PARAMETER (variable1=value, variable2=value, ...)
```

- Example: value of π

```
        PARAMETER (PI=3.1415)
```

PARAMETER and DATA Statements

- DATA statement is used to assign initial values to a variable
- Whereas PARAMETER statement assign *permanent* values, DATA statement assigns *temporary* values
- Most useful to *replace* READ statements at the beginning of a program
- General form of array declaration

```
DATA var1,var2, .../ val1, val2, .../
```

If arrays are specified

```
DATA (array(subs),subs=start,stop,step)/val1, val2, .../
```

PARAMETER and DATA Statements

- Example:
  Without DATA statements

```
VOLTS = 5.3
RESIST = 1000.0
CAPICT = 0.000035
```

  With DATA statement

```
DATA VOLTS, RESIST, CAPICT /5.3, 1000.0, 0.000035/
```

- Turn Example 6.19–6.20 into complete programs, compile and run them to study various ways of implementing PARAMETER and DATA statements