UNIT 11 INTRODUCTION TO DATA STRUCTURES: ARRAY

Structure **Page No.**

11.1 INTRODUCTION

This Unit is the introductory unit on data structures. In Sec. 11.2, we introduce you to program analysis and the concept of computational complexity. Since these topics are studied in the Design and analysis of algorithms course we will not go into great details. In Sec. 11.3, we will start our study of data structures with the simplest data structure, namely Array. You have already seen arrays in Unit 6 and Unit 7 of Block 2. In this unit, we are going to look at this array as a data structure. In Sec. 11.4 of this Unit, we will see how to create arrays and perform some elementary operations on them. In Sec 11.5, we will discuss different ways of storing data in an array like row major, column major methods. In the last section, Sec 11.6, we will discuss sparse arrays which are large arrays in which most entries are the same, usually 0.

Objectives

After studying this unit, you should be able to

- explain the benefits of program analysis;
- define a data structure;
- perform basic operations on arrays; and
- explain how data is stored in sparse arrays.

11.2 PROGRAM ANALYSIS

In this section, we introduce you to *program analysis*. What do we mean by this? After all, there are many ways of analysing a program. For instance, we can analyse a program from any of the following points of view.

i) Verifying that it satisfies the requirements

ii) Proving that it runs correctly without any logical errors.

iii) Determining if it is readable.

iv) Checking that modifications can be made easily, without introducing new errors.

v) We may also analyse program execution time and the storage complexity associated with it, i.e. how fast does the program run and how much storage it requires.

Another related question can be : How big must its data structure be and how many steps will be required to execute its algorithm?

Since this course concerns data representation and writing programs, we shall analyse programs in terms of storage and time complexity.

Performance Issues

In considering the performance of a program, we are primarily interested in

i) how fast does it run?

ii) how much storage does it use?

Generally we need to analyse efficiencies, when we need to compare alternative algorithms and data representations for the same problem or when we deal with very large programs.

We often find that we can trade time efficiency for space efficiency, or vice-versa. For finding any of these i.e. time or space efficiency, we need to have some estimate of the problem size. Let's assume that some number N represents the size of the problem. The size of the problem or N can reflect one or more features of the problem, for instance N might be the number of input data values, or it is the number of elements of an array etc. In the case of algorithms that take integers as input like the algorithm to find the hcf, it is the number of digits in the input. For example, N may be the number of names of persons that we want to sort in alphabetical order.

Let us consider an example here. Suppose that we are given two algorithms for finding the largest value in a list of N numbers. It is also given that second algorithm executes twice the number of instructions executed by the first algorithm for each N value.

Let first algorithm execute S number of instructions. Then second algorithm would execute 2S instructions. If each instruction takes 1 unit of time, say 1 millisecond then for N = 10, 100, 1000 and 10000 we shall have the number of operations and estimated execution time as given below:

N	Algorithm I		Algorithm II	
	Number of Instructions	Estimated Execution Time	Number of Instructions	Estimated Execution Time
10	10S	10 msec	20S	20 msec
100	100S	100 msec	200S	200 msec
1000	1000S	1000 msec	2000S	2000 msec
10000	10000S	10000 msec	20000S	20000 msec

1 millisec = 1 msec = 1/1000 sec

You may notice that for larger values of N, the difference between execution time of the two algorithm is appreciable, and one may clearly say that Algorithm II is slower than Algorithm I. Also, as the problem size becomes larger and larger, Algorithm I performs better and better than Algorithm II. This kind of performance improvement is termed as *order of improvements*. Two algorithm may compare with each other by a constant factor, i.e. improvement from one to another does not change as the problem size gets larger. For example, one of them can be two times faster than the other and it always remain two times better regardless of the problem size.

In both the above algorithms, the time taken grows linearly with input size. Actually, instead of time, we usually talk in terms of the number of basic operations the algorithm has to perform as a function of the size of the input. For example, to add two n digit integers, the computer has to perform n operations. We have to add each digit in the first number to the corresponding digit in the second number. In this case, the number of operations grows linearly with the size of the input. On the other hand, the usual algorithm for multiplying two n digit numbers performs about n^2 operations. This is because, under the usual multiplication algorithm, we have to multiply each digit of the first number with the each digit of the second number. See Fig. 1 for

the number of operations required to add and multiply 3 digit numbers.

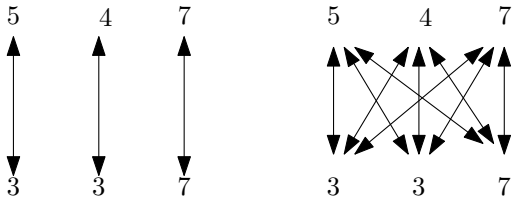

Fig. 1: Number of operations for adding and multiplying two 3 digit numbers.

Therefore, the number of operations required to multiply two n-digit numbers grow as the square of the size of the numbers. Such algorithms are called *quadratic algorithms*. What we are actually doing is to compare the growth rate of algorithms with known functions; in the above two cases we see that the growth is comparable to a linear polynomial and a quadratic polynomial, respectively.

There is a standard notation for such comparisons called the big-Oh notation. If $f(n)$ and $g(n)$ are two functions defined on natural numbers, we say that $f(n) = O(g(n))$ if $|f(n)| \leq K|g(n)|$ where K is a constant independent of n. If $f(n)$ is the number of operations an algorithm has to perform on an input of size n, if $f(n) = O(n)$, we say that the algorithm is linear algorithm; if $f(n) = O\left(n^2\right)$, we say that the algorithm is a quadratic algorithm. More generally, if $f(n) = O(g(n))$ for some polynomial g(n), we say that the algorithm is a polynomial time algorithm. We discuss the analysis(and design) of algorithms in the Design and Analysis of Algorithms course.

Average Case and Worst Case Analysis

An algorithm chooses an execution path depending on the set of data values (input). Therefore, an algorithm may perform differently for two different sets of data values. If we take a set of data values for which the algorithm takes the longest possible execution time, it leads us to the worst case execution time. On the other hand, an average case execution time is the execution time takes by algorithm for an expected range of data values. For analysis of an algorithm to predict its average or worst case execution time, we need to make certain assumption such as assuming that all operations take about the same amount of time.

For example, suppose we have n student cards with the enrolment numbers a_1, a_2, …, a_n written on them and they are in a box. A number is called out and you have to take out the card with that number. Let us assume that the cards are arranged in ascending order, i.e. that is a_1 is the topmost card, a_n is the card at the bottom and $a_i < a_{i+1}$ for $1 \leq i \leq n-1$. If you do not know this and you have to search through the cards and find the k^{th} card, you have to check through k cards. So, the number of operations you need depends on the card you want to find. In the worst case, if you want to find the n^{th} card, you have to search through n cards.

What is the *average time* you take to find any card? Suppose that all the cards are equally probable to be called. Then, the probability of any card being called out is $\frac{1}{n}$ and the average number of cards you have to search through is $\sum_{i=1}^{n}(i) \cdot \frac{1}{n}$ since the number of cards you have to search through to find the i^{th} card is i. The sum is

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

One issue we have to deal with when we want to optimise the performance is the representation of data. We have find a representation that optimises the time(or space) involved in working with the data. We use **Data structures** for this. We do not merely want to represent data, but also work with them. So, our data structures should allow us to perform the required operations in an efficient way. So, we can say that

$$\text{Data Structures} = \text{Organised Data} + \text{Allowed operations}$$

In the next section, we begin our study of data structures with arrays, the simplest of data structures.

11.3 ARRAYS AS DATA STRUCTURES

In applications where we have a small number of items to handle, we tend to specify separate variables names for each item. When we have to keep track of more pieces of related data, we need to organise data, in such a way that we can use one name to refer to several items. Let us see this through a simple example. Consider the following problem:

Read 25 numbers and print them in reverse order.

The problem requires all the numbers as they are read. Further we cannot print anything until all 25 numbers are read; therefore, we need to store all the twenty five numbers. Reading 25 numbers in 25 different variables will be quite cumbersome and so would be writing these numbers in reverse order. It is much simpler to call the numbers NUM_1, NUM_2, NUM_3, $\cdots$, NUM_{25}.

Each number is a NUM and numbers are distinguished by subscripts. Also, they are read in succession. Thus, we can abbreviate this sequence as NUM_i for $i = 1, 2, \cdots, 25$. Such a subscripted variable is called an **array**. More formally, an array is a finite, ordered set of homogeneous elements which are stored in adjacent cells in memory. Arrays are usually used when a program includes a list of recurring elements.

You are probably wondering 'What is new about arrays? We have discussed them already in Block 2.'. Here, we are going to study array as an abstract object without reference to any language. The set of integers exists independently of any representation or implementation. In this set you can carry out basic operations like addition, subtraction, multiplication and division. The C data type **int** is a particular implementation of this. It allows you to carry out all the basic operations like addition, multiplication etc. This is not the full set of integers, but only integers between -32767 and 32767 according to C89 specification. Similarly, we can think of an abstract data type called array with a specified set of operations.

One characteristic feature of the array is that it takes the same amount of time to access any element in the array. As we have seen, in C, elements of an array can be accessed using subscripts placed in square brackets[]. Repetition over a sequence of values of i may also be implemented using a loop construct. For example, the following statement reads all 25 values:

```
for(i = 0;i < 25; ++i)
    scanf("%d",NUM[i]);
```

A similar approach works out for printing the values.

The simplest form of an array is a one-dimensional array or vector. As stated earlier, the various elements of an array are distinguished by giving each piece of data separate index or subscript. The subscript of an element designates its position in array's ordering. An array named A which consists of N elements can be depicted as shown in Fig. 2.

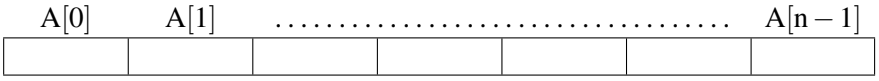

Fig. 2: One dimensional array.

Arrays can be multi-dimensional. Any array defined to have more than one dimension is considered to be multi-dimensional array. An array can be 2-dimensional, 3-dimensional, 4-dimensional, or N-dimensional although they rarely exceed three dimensions.
Two-dimensional arrays, sometimes called matrices, are quite common. The best way to think about a two-dimensional array is to visualise a table of columns and rows: the first dimension in

the array refers to the rows, and the second dimension refers to the columns. Let us see an example of a 2-dimensional array.

A collection of data about the grades of students in a class in the four different exams can be represented using a 2-dimensional arrays. If we have 10 students and each given grades in 4 exams, we can depict it as in Fig. 3. Each cell in this table contains a grade value for the student

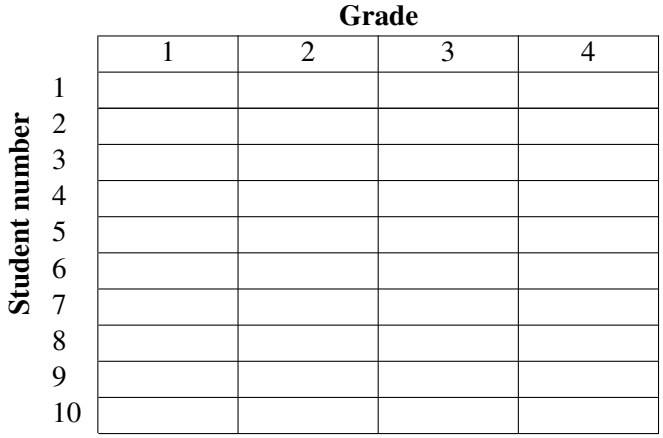

Fig. 3: An example of a two dimensional array.

Number (given by the corresponding row number) and exam number (given by the corresponding column no.). We may map it on to an array A of order 10×4. `A[I][J]` represents an element of A, where I runs from 1 to 10 and J runs from 1 to 4. `A[3][4]` will have the grade value of 3rd student in fourth exam, `A[8][1]` will have the grade value of 8th student in first exam, and so on.

In C, we know that the row index runs from 0 to 3 and the column index runs from 0 to 9. More about this later.

By convention, the first subscript of a 2-dimensional array refers to a row of the array, while the second subscript refers to a column of the array.

In general, an array of the order $M \times N$ (read as M by N) consists of M rows, N columns and MN elements. It may be depicted as in Fig. 4 Let us now discuss the syntax and semantics of an

0	1	.....	N – 1
1			
⋮			
⋮			
M – 1			

Fig. 4: A 2 dimensional array.

array. We can divide our discussion in three parts:

- Array declaration
- Storage of Arrays in Main Memory
- Use of Arrays in Programs

In the next section, we will discuss creation of arrays and elementary operations that can be performed on arrays.

11.4 CREATION OF ARRAYS AND ELEMENTARY OPERATIONS

Three things need to be specified to declare an array in most of the programming languages:

- the array name
- the type of data to be stored in array elements
- the subscript range

In C language the array declaration is a follows:

```
int A[24];
float B[100][25];
```

In first declaration A is the array name; the elements of A can hold integer data and the number of elements is 24 i.e. subscripts range from 0 to 23.

In the next declaration B is the array name; the data type of its elements is real and it is a 2-dimensional array with subscripts ranging from 0 to 99 and 0 to 24. In other languages, the lower limit on an array does not have to be 1. It makes more sense to start the array at a value that corresponds to the context of your data. Also, subscript need not always be positive in some languages. It can be negative or zero. However, not all programming languages allow zero or negative subscripts.

Be careful when using arrays with indexes beginning with 0. Failing to remember that the zero elements is the first item in the array — and therefore, the element at index 5 is the sixth, not the fifth — is a frequent cause of programming bugs.

You can begin your indexes at 0 or at 1 or at any other value, if programming language in use allows it. There is no technical reason to use one method over the other. However most programmers prefer to start arrays at 0 — even though it is easier to begin at 1. The reason for this is that some languages — C and C++ for example — require arrays to begin with zero indexes. If you define most of your arrays the same way, your programs will be easier to convert to these languages.

An array declaration tells the computer two major pieces of information about an array. First, the range of subscripts allow the computer to determine how many memory locations must be allocated. Second the array type tells the computer how much space is required to hold each value. Let us consider the following declarations:

```
int A[10];
float B[10];
```

The first declaration tells the computer to allocate enough space for the variable A to store 10 integers. The second declaration tells the computer to allocate enough space for the variable B to store 10 reals. Since a real number takes more space than an integer the storage allocated would not be same. We have already discussed declaration of arrays in C in Units 12 and 13 of Block 2.

Operations on Arrays

The array is a *homogeneous structure*, i.e. the elements of an array are of the same type. It is *finite*; it has a specified number of elements. An array is *ordered*; there is an ordering of elements in it as zeroth, first, second etc. Following set of operations are defined for this structure.

i) Creating an array

ii) Initialising an array

iii) Storing an element

iv) Retrieving an element

v) Inserting an element

vi) Deleting an element

vii) Searching for an element

viii) Sorting elements

ix) Printing an array

Let us now write a function that deletes an element from an array. The function in Listing 1 takes the name of the array(which is actually a pointer to an **int**), the index of the element to be removed and the index of the last element as arguments. It removes the element and returns the index of the last element.

```
int delete_element(int *list, int last_index, int index)
{
    int i;
    for(i = index; i < last_index; i ++)
        list[i]=list[i+1];
    return (last_index-1);
}
```

Listing 1: A function to delete an element from an array.

Let us now see how to insert an element in an array. The function in Listing 2 takes the name of the array, the element to be inserted, the index of the last element and the postion where the element has to be inserted.

```
int insert_element(int *list, int num, int last_index,
int index)
{
    int i;
    if ( last_index == max_array_size){
        Error("Array is full. Cannot insert element.");
        return (-1);
    }
    else {
        for ( i = last_index-1; i > index; i--)
            list[i+1]=list[i];
        list[index]=num;
   }
        return(last_index+1);
}
```

Listing 2: A function to insert an element in an array.

Here is an exercise to check you understanding of our discussion so far.

E1) Write a program that

1) creates an array consisting of elements 3, 4, 5, 6 and 7;
2) removes 5 from the array;
3) inserts 10 after 7 in the array.

We will see how to sort an array of integers using **insertion sort.**

Example 1: Suppose we want to sort the list 21, 20, 19, 23, 16, 25. We do this in 5 **passes**. In the first pass, we look at 20 which is less than 21, yet it appears before 20. So, we exchange 20 and 21. In the second pass, we look at the third element in the list, which is 19. We see that the two elements before it, 21 and 20, are both bigger. So, we move 21 and 20 right by one position and insert in the position occupied by 21 previously. Now, the first three elements are in the correct order. We proceed like this to sort all the elements of the list. In general, in the i^{th} pass, insertion sort ensures that all the $i+1$ elements from postions 0 through i are in sorted order. In

Table 1: Interchanges in pass 2.

20	21	19	23	16	25
20	19	21	23	16	25
19	20	21	23	16	25

Table 2: Insertion sort.

Initial	21	20	19	23	16	25
Pass 1	20	21	19	23	16	25
Pass 2	19	20	21	23	16	25
Pass 3	19	20	21	23	16	25
Pass 4	16	19	20	21	23	25
Pass 5	16	19	20	21	23	25

the i^{th} pass, we move the $(i+1)^{th}$ element till its correct place. The function Listing 3 finds the correct position of the $(i+1)^{th}$ element in the i^{th} pass:

```
int find_position(int *list, int pos)
{
    int j;
    for (j = pos; j > 0 && list[j - 1] > list[pos]; j--);
    return (j);
}
```

Listing 3: A function that finds the correct position of an element.

After that, we have to insert the $(i+1)^{th}$ element in the correct position. We can now insert the element in the correct position using the function we wrote for inserting an element in an array.

* * *

Here is an exercise for you to check your understanding of the previous example.

E2) Write a program that scans an array of integers, sorts them by insertion sort and prints the sorted array.

E3) We have used two different functions to find the correct position of an element and for inserting the element. Both can be performed in one go. Write an insertion sort function that does this.

These array operations apply to arrays of any dimension. We have already used these operations in our programme that lists primes. Here is a program that gives examples some of these operations.

```
1  /*Program 11.3: 2-dimensional Array operations
2  example. File name: unit11-matrix-ex.c*/
3  #include<stdio.h>
4  #define num_rows 4
5  #define num_columns 4
6  void printarray(int mymat[num_rows][num_columns]);
7  void findelement(int mymat[num_rows][num_columns], int x);
8  int main(void)
9  {
10     /* Create and initialise an array of ints */
11     int i, j, mat[num_rows][num_columns] = { {1, 0, 0, 0} };
12     /*print the array */
13     printf("First call to printarray ...\n");
14     printarray(mat);
15     /* Insert elements; Make it  4x4 diagonal matrix */
```

```
16      for (i = 0; i < num_rows; i++)
17          for (j = 0; j < num_columns; j++)
18          {
19              if (i == j)
20                  mat[i][j] = 3;
21              else
22                  mat[i][j] = 0;
23          }
24      printf("\n\n");
25      printf("Second call to printarray...\n");
26      printarray(mat);
27      for (i = 0; i < num_rows; i++)
28          for (j = 0; j < num_columns; j++)
29              mat[i][j] = i + j;
30      printf("\n\n");
31      printf("Third call to printarray...\n");
32      printarray(mat);
33      printf("\n\n");
34      findelement(mat, 4);
35      return (0);
36  }
37  void printarray(int mat[num_rows][num_columns])
38  {
39      int a, b;
40      for (a = 0; a < num_rows; printf("\n"), a++)
41          for (b = 0; b < num_columns; b++)
42              printf("mat[%d][%d]=%d, ", a, b, mat[a][b]);
43  }
44  void findelement(int mat[num_rows][num_columns], int x)
45  {
46      int i, j, found;
47      found = 0;
48      printf("Searching for %d ...\n", x);
49      for (i = 0; i < num_rows; j = 0, i++)
50          {
51          for (j = 0; j < num_rows; j++)
52              if (mat[i][j] == x)
53              {
54                  found = 1;
55                  break;
56              }
57          if (found == 1)
58              break;
59          }
60      if (found == 0)
61          printf("Could not find %d", x);
62      else
63          printf("Found ! mat[%d][%d]=%d\n", i, j, x);
64  };
```

Here is the output from the program:

```
/*Output from Prog 1.1*/ First call to printarray ...
mat[0][0]=1, mat[0][1]=0, mat[0][2]=0, mat[0][3]=0,
mat[1][0]=0, mat[1][1]=0,
mat[1][2]=0, mat[1][3]=0, mat[2][0]=0, mat[2][1]=0,
mat[2][2]=0, mat[2][3]=0,
mat[3][0]=0, mat[3][1]=0, mat[3][2]=0, mat[3][3]=0,
Second call to printarray...  mat[0][0]=3,
mat[0][1]=0, mat[0][2]=0,
```

```
mat[0][3]=0, mat[1][0]=0, mat[1][1]=3, mat[1][2]=0,
mat[1][3]=0,

mat[2][0]=0, mat[2][1]=0, mat[2][2]=3, mat[2][3]=0,
mat[3][0]=0, mat[3][1]=0,

mat[3][2]=0, mat[3][3]=3,

Third call to printarray...  mat[0][0]=0, mat[0][1]=1,
mat[0][2]=2,

mat[0][3]=3, mat[1][0]=1, mat[1][1]=2, mat[1][2]=3,
mat[1][3]=4,

mat[2][0]=2, mat[2][1]=3, mat[2][2]=4, mat[2][3]=5,
mat[3][0]=3, mat[3][1]=4,

mat[3][2]=5, mat[3][3]=6,

Searching for 4 ...  Found !  mat[1][3]=4
```

In this program, line 10 declares and initialises an array of size `num_rows` $\times$ `num_columns`. Note that although the matrix has four rows, we have given only one row in the declaration. You may recall that in this case, all the remaining elements are set to 0. We can confirm this by a call to the `printarray` function in line 13 to print the values of the array.

Lines 15 to 22 convert the matrix into a 4×4 "diagonal matrix" with 3 along the diagonal. Again, the function `printarray` prints the array.

Lines 26 to 28 set the value of `mat[i][j]` to `i + j`. Again, we call `printarray` function to print the values of the array.

Here are some exercises for you to test your understanding of array operations.

E4) Write a program in C that declares a 4×4 array and reads the entries of the array from the terminal
   a) Row by row.
   b) Column by column.

E5) Write a function in C that takes an array of integers of size 4×4 as input and changes the all the elements above the diagonal to zero.

We close the section here. In the next section, we will discuss how the entries of an array are stored in the main memory.

11.5 STORAGE OF ARRAYS IN MAIN MEMORY

Let us now see how the data represented in an array is actually stored in the memory cells of the machine. Because computer memory is linear, a one-dimensional array can be mapped on to the memory cells in a rather straight forward manner. Storage for element A[I+1] will be adjacent to storage for element A[I] for $I = 1, 2, \ldots, N$. We assume that the size of each element stored is one unit. To find the actual address of an element one merely needs to subtract one from the position of the desired entry and then add the result to the address of the first cell in the sequence.

Let us look at an example. Consider an array A of 25 elements. We require to find the address of A[4]. If the first cell in the sequence A[0], A[1], A[2], ... , A[25] was at address 16, then A[4] would be located at $16 + (5 - 1) = 20$, as shown in Fig. 5 on the facing page. Therefore, it is necessary to know the starting address of the space allocated to the array and the size of the

Memory	16	17	18	19	20	···	···	···
	A[0]	A[1]	A[2]	A[3]	A[4]	···	···	···

Fig. 5: Storage of Arrays.

each element which is same for all the element of an array. We may call the starting address as a base address and denote it by B. Then the location of Ith element would be

$$B + (I - 1) * S \tag{1}$$

where S is the size of each element of array. We refer you to the fourth example program in Unit 12 of Block 2 which illustrates this.

Let us now consider storage mappings for multi-dimensional arrays. As we had seen in previous section that in a 2-dimensional array we think of data being arranged in rows and columns. However Machine's memory is arranged as a row of memory cells. Thus the rectangular structure of a 2-dimensional array must be simulated. We first calculate the amount of storage area needed and allocate a block of contiguous memory cells of that size. One way to store the data in the cells is row by row. That is, we store first the first row of the array, then the second row of the array and then the next and so on. For example, the array defined by A which logically appears as given in Fig. 5 appears physically as given in Fig. 7. Such a storage scheme is called Row Major Order.

The other alternative is to store the array column by column. It is called Column Major Order. The array of Fig. 8 on the next page shows the physical arrangement in Column Major order.

E6) Create a two-dimensional array whose number of rows are 10 and columns are 26 and the component type is character.

E7) Show how the array

$$\begin{matrix} 1 & 3 & 7 \\ 5 & 2 & 8 \\ 9 & 7 & 1 \end{matrix}$$

would appear in the memory when stored in

i) Row major order

ii) Column major order

$$\begin{bmatrix} A[0][0] & A[0][1] & A[0][2] & A[0][3] \\ A[1][0] & A[1][1] & A[1][2] & A[1][3] \\ A[2][0] & A[2][1] & A[2][2] & A[2][3] \end{bmatrix}$$

Fig. 6: Logical representation.

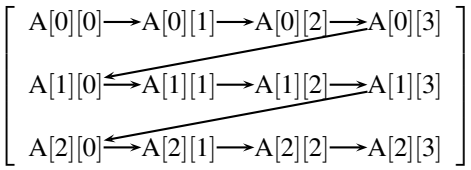

Fig. 7: Row major representation.

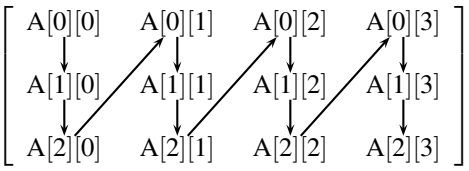

Fig. 8: Column Major Representation.

In all implementations of C the storage allocation scheme used is the Row Major Order.

Let us now see how do we calculate the address of an element of a 2-dimensional array, which is mapped in Row Major Order. Consider a 4×6 array `A[4][6]`. Take B as the array's base address and S as the size of element of the array.

Remember that in C, array indices start with 0. So, to locate element `A[I][J]` we must skip I rows$(0, 1, 2, \ldots, I-1)$; each having 6 elements, each element of length S and (J) elements of Ith row, each of length S. Therefore, the address of element `A[I][J]` would be

$$B + I \cdot 6 \cdot S + J \cdot S \tag{2}$$

We may now generalise this expression for a 2-dimensional array

$$A[U_0, U_1]$$

where $U_0 - 1$ and $U_1 - 1$ are the upper bounds of the two subscript ranges.

The location of an element `A[I][J]` for such an array would be

$$B + I * U_1 * S + J * S$$

The row major order varies the subscripts in right to left order. For example, the elements of a 2-dimensional array $A[U_1, U_2]$ would be stored in following order:

$A[0,0]$
$A[0,1]$
$\vdots$
$A[0, U_2 - 1]$
$A[1,0]$
$A[1,1]$
$A[1,2]$
$\vdots$
$A[1, U_2 - 1]$
$\vdots$
$A[U_1 - 1, U_2 - 1]$

We may generalise it for an N-dimensional array $A[U_0]A[U_1] \ldots A[U_n - 1]$. The elements would be stored in following order:

$A[0][0] \ldots [0]$
$A[0][0] \ldots [1]$
$\vdots$
$A[0][0] \ldots A[0]A[U_n - 1]$
$A[0][0] \ldots A[1]A[0]$
$\vdots$
$A[0][0] \ldots [1][1]$
$\vdots$
$A[U_0 - 1]A[U_1 - 1] \ldots A[U_n - 1]$

Let us see how the above expressions work out for a column major order.

We once again consider a 4×6 array A[4][6]. Also take B as base address and S as size of each element. Then the address of A[I][J] would be

$$B + J \cdot 4 \cdot S + I \cdot S$$

To reach A[I][J] we shall skip J-1 columns, each of length $4 * S$ and I-1 elements each of length S.

Let us now generalise it for an array $A[U_1, U_2]$.

Following the same logic, the address of A[I][J] would be given as

$$B + J \cdot U_1 \cdot S + I \cdot S$$

The column major order varies the subscripts in left to right order. For example the elements of a 2-dimensional array A[4][3] would be stored in the sequence as given below:

A[0][0]
A[1][0]
A[2][0]
A[3][0]
A[0][1]
A[1][1]
A[2][1]
A[3][1]
A[0][2]
A[1][2]
A[2][2]
A[3][2]

E8) How would a 4×3 array A[4][3] stored in Row Major Order?

E9) How would a $m \times n$ array A[m][n] stored in Column Major Order?

As we had done for Row Major Order, we may generate the sequence of N-dimensional array

$A[U_1][U_2][U_3]\ldots[U_n]$

as stored in Column Major Order. It would be as given below:

$A[0][0]\ldots[0]$
$A[1][0]\ldots[0]$
$\vdots$
$A[U_1 - 1][0]\ldots[0]$
$A[0][1]\ldots[0]$
$\vdots$
$A[0][2]\ldots[0]$
$\vdots$
$A[0][U_2 - 1]\ldots[0]$
$\vdots$
$A[U_1 - 1][U_2 - 1]\ldots[U_n - 1]$
$\vdots$

In the numerical computations involving matrices, we often come across matrices where we need to deal with matrices that have lots of zeros. We will be wasting computing space as well

as time if we use the usual methods for working with these arrays. We will look at some special methods for representing and working with these matrices.

11.6 SPARSE ARRAYS

Often, in numerical computing, we come across matrices with lots of zeros. Such matrices are called **sparse matrices**. Sparse arrays are special arrays which arise commonly in applications. It is difficult to draw the dividing line between the sparse and non-sparse array. Loosely an array is called sparse if an entry(commonly 0) occurs relatively large number of times. For example, in Fig. 9, out of 49 elements, only 6 are non-zero. This is a sparse array. If we store those array

$$\begin{array}{ccccccc} 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 \end{array}$$

Fig. 9: A sparse array.

through the techniques presented in previous section, there would be much wasted space.

Let us consider 2 alternative representations that will store explicitly one the non-zero elements.

1) Vector representation
2) Linked List representation

We shall discuss only the first representation here in this Unit. The Linked List representation shall be discussed in a later Unit.

Each element of a 2-dimensional array is uniquely characterised by its row and column position. We may, therefore, store a sparse array in another array of the form.

$$A[n+1][3]$$

where n is number of non-zero elements.

The sparse array given in Fig. 9 may be stored in the array A[7][3] as shown in Fig. 10. The

$$A = \begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 0 & 7 & 7 & 6 \\ 1 & 1 & 6 & 1 \\ 2 & 2 & 5 & 1 \\ 3 & 4 & 5 & 3 \\ 4 & 5 & 2 & 2 \\ 5 & 6 & 5 & 4 \\ 6 & 7 & 5 & 2 \end{array}$$

Fig. 10: Sparse array representation using vector representation.

Elements A[0][0] and A[0][1] contain the number of rows and columns of the sparse array. A[0][2] contains the number of non-zero elements of sparse array. The first and second element of each of the rows store the number of row and column of the non-zero term and the third element stores the value of non-zero term. In other words, each non-zero element in a

2-dimensional sparse array is represented as a triplet with the format (row subscript, column subscript, value).

If the sparse array was one-dimensional, each non-zero element would be represented by a pair. In general for an N-dimensional sparse array, non-zero elements are represented by an entry with N+1 values.

If you are interested in alternate representations of sparse matrices and methods for working with them, you may refer to Sec. 2.7 in the book *'Numerical recipes in C'* and the references given there.

11.7 SUMMARY

In this Unit, we saw

i) the benefits of program analysis.

ii) the definition of a data structure

iii) how to perform basic operations on arrays

iv) how data is stored in sparse arrays

11.8 SOLUTIONS/ANSWERS

E1)
```
/*Program 11.1. A simple example of Array operations
File name:unit11-insert-element.c*/
#include<stdio.h>
#define max_array_size 20
void Error(char *message);
void Message(char *message);
int printarray(int *list,int limit);
int delete_element(int *list, int last_index, int index);
int insert_element(int *list, int num, int last_index, int index);
int main()
{
    int a[max_array_size]={3,4,5,6,7}, last = 5;
    printarray(a,last);
    last = delete_element(a,last,2);
    Message("After deleting 5 the array is:");
    printarray(a,last);
    last = insert_element(a,10,last,4);
    Message("After inserting 10 in the fifth position \
the array is:");
    printarray(a,last);
    return (0);
}
void Error(char *message)
{
    printf("\n");
    fprintf(stderr,"Error! %s\n",message);
}
void Message(char * message)
{
    fprintf(stdout,"\nMessage: %s\n",message);
}
int printarray(int *list,int last)
{
    int i;
    for(i = 0; i < last; i++)
```

```
        printf("%d\n", list[i]);
    return (0);
}
int delete_element(int *list, int last_index, int index)
{
    int i;
    for(i = index; i < last_index; i ++)
        list[i]=list[i+1];
    return (last_index-1);
}
int insert_element(int *list, int num, int last_index,
int index)
{
    int i;
    if ( last_index == max_array_size){
        Error("Array is full. Cannot insert element.");
        return (-1);
    }
    else {
        for ( i = last_index-1; i > index; i--)
            list[i+1]=list[i];
        list[index]=num;
   }
        return(last_index+1);
}
```

E2)

```
/*Program 11.2. A example to show sorting of arrays.
Insertion sort. File name:unit11-insortn.c*/
#include<stdio.h>
#define max_array_size 20
int find_position(int *a, int num);
int insert_number(int *a, int cpos, int npos, int num);
int main()
{
    int i, k = 0, array[max_array_size] = { 0 }, list_size;
    printf("Enter the size of the list:\n");
    scanf("%d", &list_size);
    printf("Enter the elements of the array:\n");
    for (i = 0; i < list_size; i++)
        scanf("%d", &array[i]);
    for (i = 1; i < list_size; i++) {
        k = find_position(array, i);
        if (k != i)
            insert_number(array, i, k, array[i]);
    }
    printf("The sorted array is:\n");
    for (i = 0; i < list_size; i++)
        printf("%d\n", array[i]);
    return (0);
}
int find_position(int *list, int pos)
{
    int j;
    for (j = pos; j > 0 && list[j - 1] > list[pos]; j--);
    return (j);
}
int insert_number(int *list, int cpos, int npos, int num)
/* cpos is the current position of num;
npos is the new position of num*/
{
    int i;
/*Shift elements to the right by one place*/
    for (i = cpos; i > npos; i--)
```

```
        list[i] = list[i - 1];
    /*Insert the number at the new position.*/
        list[npos] = num;
        return 0;
    }
```

E3)
```
    int insertion_sort(int *list, int LL)
    {
        int cp, j, temp;
        for (cp = 1; cp < LL; cp++)
            {
            temp = list[cp];
            for (j = cp; j > 0 && list[j - 1] > temp; j--)
                list[j] = list[j - 1];
            list[j] = temp;
            }
        return (0);
    }
```