exhausted. In another approach, the words of the dictionary may be stored in alphabetical order similar to the manner in which words are found in the dictionary itself. In this approach, searching becomes easier because it can exploit the ordering of the words in the array. Specifically, one can go to the middle of the array and by comparing the given word and the word at middle one can choose only one half of the array for further inspection. This process can be continued until the middle word matches with the given word or the array is exhausted by the process of halving the array. Comparing these two approaches, it is clear that the search algorithm is fundamentally different in these two approaches. Consequently, complexity of the algorithms is also different. In yet another approach, in addition to keep the words arranged alphabetically, another list is maintained to remember the position of the first word starting with a specific letter. That is, the second list maintains the position of the first word starting with 'A', 'B', 'C' and so on. This approach is called "indexing". The search algorithm can find out the first letter of the given word and find the part of the first array of all words starting with that letter. On this part of the array search may proceed in the way outlined in the second approach. The third approach may be efficient in many cases but it uses more space to do so. The extra space is required for storing the second list. The above discussion explains that the design of the "search" procedure is dependent on the organization of the data on which "search" works. So, it is clear that the design of an algorithm is greatly influenced by the organization of the data on which the algorithm operates.

Thus, data, its representation, organization, and operations on data are equally critical to the study of computer science as crucial the study of algorithm is. One objective of this book is to study many different types of organizations of data. For each such organization, some relevant operations will also be explored. Algorithms to implement such operations will be discussed and often the complexity of these algorithms will be computed. After understanding the topics discussed in this book, the reader should be confident on the usage and implementation of the standard organizations of data. More importantly, the reader should be able to devise new organizations of data for different applications so that the problem at hand can be solved in a more efficient way.

1.2. Abstract Data Type (ADT) and Data Structure

"Type" or "data type" is a common terminology in mathematics and in programming languages. Variables are declared to be of a particular "data type". These declarations mean that the variables can assume values only from a particular set. The type of the variable determines this set. Thus, "type" may be defined as a set of values. (However, this definition is not complete as will be seen in the sequel). There are plenty of examples of "data type" in programming languages. In lan-

guages like FORTRAN, Pascal, C, the standard built-in types include integers, real numbers, characters and so on. Variables declared of type integer may assume any integral value. However, depending on the machine, there is a limit to the range of integers that can be properly used in any programming language. Hence, there is a subtle distinction between "type" in mathematics and "type" in computers. For example, the type "integer" is definitely a finite set in any programming language. This distinction is more pronounced for real numbers. Since each number is to be finally represented in a finite number of bits in a computer, the continuity property of real numbers in mathematics is lost in the corresponding type in a programming language. In fact the set of real numbers in mathematics does not have a one to one mapping to the set of real numbers in a programming language. Moreover, a type is not just a set of values. A "type" also determines the set of operations applicable to the set of values that the said "type" represents. Note that the operations are defined while "type" itself is being defined. In other words, to define a type, the set of values and the set of operations are defined. Any object of that type can assume the values defined for that type and the operations defined on that type may be applied on the object of that type. Thus, for type "integer", the operations such as addition, subtraction, multiplication etc. are defined. These operations are defined to be binary and they are defined on two objects of type integers. So, two variables of type integer can be added. But, two variables of type character can not be, usually, added or multiplied. Although most programming languages support some standard "data types", they are not enough for most of the applications. So, programming languages usually support creating new data types in terms of "Structured types". A "Structured type" is made up of several components each of which are either an atomic type or another "Structured type". This can be accomplished in "C" language by defining a "struct". However, just by defining a structured type, the operations applicable to objects of that type can not be specified. This is why the concept of an Abstract Data Type (ADT) is required.

An ADT is defined to be a mathematical model of a user-defined type along with a collection of all primitive operations on that model. To illustrate this notion, define a new data type "SET", which denotes a set of entities. An object of type "SET" is defined to be a collection of "N" otherwise unrelated entities. If the type of each entity is represented by a set of values "D" then an object of type "SET" having "n" entities may assume any value from the set D(i.e. D X D X D X ... X D) where multiplication in this context means set multiplication. The primitive operations that may be defined for this ADT are given in the following:

1. Assign (SET A, SET B)

   This operation assigns the set identified by "B" to "A".

2. Null (SET A)

   This operation assigns null (an empty set) to "A".

3. SET Union (SET A, SET B)

= 0. Then it can be shown by induction that $F(n) \leq \phi^{n-1}$ for $n > 0$.

The induction base is true as $F(1) = 1 \leq \phi^0$. By induction hypothesis assume that the result $F(n) \leq \phi^n$ for all "k", $k < m$. To prove the result for $k = m$, note that

$$F(m) = F(m-1) + F(m-2)$$

That is, $F(m) \leq \phi^{m-2} + \phi^{m-3}$ (By applying induction hypothesis on F(m-1) & F(m-2)).

That is, $F(m) \leq \phi^{m-3}(\phi + 1)$

That is, $F(m) \leq \phi^{m-3}.\phi^2$

That is, $F(n) \leq \phi^{m-1}$.

Hence $F(n) \leq \phi^{n-1}$ for $n > 0$.

Two points have been highlighted in the analysis of the above algorithms. First, the algorithm in its entirety is not taken up for complexity analysis. Instead, some key operations, which will reveal the computational complexity of the algorithm, are considered. Second, many mathematical tools including mathematical induction are to be rigorously used to carry out the complexity analysis of algorithms.

1.4. Asymptotic Notation

It has already been mentioned that the function to measure time complexity of an algorithm is meant to be independent of machine architecture or programming language. The focus of such metrics should solely be on the order of magnitude of the frequency of execution of statements. O-notation is one of the very famous mathematical tools available for this purpose.

$f(n) = O(g(n))$ (should be read as "f of n is equal to big-oh of g of n") if and only if there are two positive constants "c" and "n" so that the following inequality holds for all $n \geq n_0$

$$|f(n)| \leq c|g(n)|$$

Try to understand the meaning of f(n). Assume that f(n) stands for computing time of some algorithm when the algorithm is run on an input of size "n". And g(n) be a known standard function like n^2, n^3, nlogn etc. That is, the behaviour of g(n) is known before the time complexity of the algorithm is computed. If analysis leads to the result $f(n) = O(g(n))$ then it means that if the algorithm is run on some computer on some input data for sufficiently large values of "n" then the resulting time will be less than some constant times $|g(n)|$.

The O-notation has been extremely useful to classify algorithms by their performances. This notation also helps designers to search for the "best" algorithms for some problems. If it can be shown that the complexity of an algorithm is O(f(n)) and there does not exist any algorithm with complexity O(g(n)) such that g(n) is weaker than f(n) then, the algorithm is said to be of optimal complexity. However, the results expressed using O-notation must be interpreted very carefully. The following points must be kept in mind in this context.

- Complexity expressed in O-notation is only an upper bound and the actual complexity may be much lower.
- This complexity can almost be treated as worst case complexity. But the input that causes the worst case may be unlikely to occur in practice.
- The constant c is unknown and is not necessarily small.
- Similarly the constant n_0 is unknown and may not be small.

It is obvious that merely stating that the complexity of an algorithm is O(f(n)) does not mean that the algorithm ever takes that long time. It just means that the analysis proves that the algorithm does not take a longer time. Actual running time required in practice might be much lower. This calls for developing better notation where it is also known that there is some input for which the algorithm actually takes O(f(n)) time.

Worst case complexity is not always very helpful as well. It may so happen that although the worst case complexity is O(f(n)), typical input data is never like that. Therefore, the average case complexity of the algorithm is much less than its worst case complexity. A very classic example is the "Quick Sort" algorithm. Although its worst case complexity is $O(n^2)$, its average case complexity is O(nlogn).

The constants "c" and "n_0" implicit in the O-notation hide the details of implementation. But these constants are actually important when the algorithms are executed in practice. See the following figure for an illustration. Although $O(n^{3/2})$ will be assumed to be larger than $O(n\log^2 n)$, that is not the case for small values of "n". Even when "n" runs into thousands, $n\log^2 n$ is more than $n^{3/2}$. This remains the case until "n" runs into tens of thousands. See Figure 1.2.

n	¼ nlogn	½ nlogn	$n\log^2 n$	$n^{3/2}$
10	22	45	90	30
100	900	1800	3600	1000
1000	20250	40500	81000	31000
10000	422500	845000	1690000	1000000
100000	6400000	12800000	25600000	31600000
1000000	90250000	180500000	361000000	1000000000

Figure 1.2 Effect of constants on f(n).

It is extremely difficult to derive the complete formula for time complexity. Most often only the leading term is calculated. Thus, in most cases, the order of magnitude of complexity is the only result that can be achieved. The values of the constants are not known at that time. It is not usual to calculate the values of the constant times unless they greatly influence the behaviour of the algorithm.

- Complexity expressed in O-notation is only an upper bound and the actual complexity may be much lower.
- This complexity can almost be treated as worst case complexity. But the input that causes the worst case may be unlikely to occur in practice.
- The constant c is unknown and is not necessarily small.
- Similarly the constant n_0 is unknown and may not be small.

It is obvious that merely stating that the complexity of an algorithm is O(f(n)) does not mean that the algorithm ever takes that long time. It just means that the analysis proves that the algorithm does not take a longer time. Actual running time required in practice might be much lower. This calls for developing better notation where it is also known that there is some input for which the algorithm actually takes O(f(n)) time.

Worst case complexity is not always very helpful as well. It may so happen that although the worst case complexity is O(f(n)), typical input data is never like that. Therefore, the average case complexity of the algorithm is much less than its worst case complexity. A very classic example is the "Quick Sort" algorithm. Although its worst case complexity is $O(n^2)$, its average case complexity is O(nlogn).

The constants "c" and "n_0" implicit in the O-notation hide the details of implementation. But these constants are actually important when the algorithms are executed in practice. See the following figure for an illustration. Although $O(n^{3/2})$ will be assumed to be larger than $O(nlog^2n)$, that is not the case for small values of "n". Even when "n" runs into thousands, $nlog^2n$ is more than $n^{3/2}$. This remains the case until "n" runs into tens of thousands. See Figure 1.2.

n	¼ nlogn	½ nlogn	$nlog^2n$	$n^{3/2}$
10	22	45	90	30
100	900	1800	3600	1000
1000	20250	40500	81000	31000
10000	422500	845000	1690000	1000000
100000	6400000	12800000	25600000	31600000
1000000	90250000	180500000	361000000	1000000000

Figure 1.2 Effect of constants on f(n).

It is extremely difficult to derive the complete formula for time complexity. Most often only the leading term is calculated. Thus, in most cases, the order of magnitude of complexity is the only result that can be achieved. The values of the constants are not known at that time. It is not usual to calculate the values of the constant times unless they greatly influence the behaviour of the algorithm.

The most common computing times of algorithms are

$$O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2) < O(n^3) < O(2^n)$$

logn If the time complixity of an algorithm is logn, then the algorithm is said to be logarithmic. When a program is logarithmic, the program becomes slightly slower as "n" increases. This time complexity usually occurs in algorithms which solves big problems by transforming it into smaller ones and thereby cutting the size of the input by some fraction. It is also interesting to note that a change in the base of the logarithm results in a change in the constant term of the complexity and that change is also very small. When "n" is 1000, logn is 3 if the base is 10 and logn is approximately 10, if the base is 2. When n is a million, logn is doubled. Whenever n becomes two-fold, logn only increases by a constant.

n When the time complexity of an algorithm is "n", then the algorithm is said to be linear. Usually this is the case, when some processing or the other has to be carried out on each individual element of the input. Naturally the time required by an algorithm is linearly proportional to "n". Thus, if "n" is doubled, the processing time of the algorithm is also doubled. This is the complexity of the optional solution for an algorithm if the algorithm must process all "n" inputs or must produce "n" outputs.

nlogn If an algorithm solves a problem by breaking it up into smaller sub-problems and solving these sub-problems independently and finally combining the solutions to the sub-problems then, the time complexity of such algorithms turns out to be nlogn.

n^2 If time complexity of an algorithm is n^2, then the algorithm is said to be quadratic. Since n^2 increases very rapidly with an increase in "n", quadratic algorithms are practically useful only for relatively small problems. When an algorithm processes all pairs of data item then, the time complexity of the algorithm turns out to be quadratic.

n^3 If an algorithm processes triplets of data items, the complexity of the algorithm becomes cubic and can not be practically useful.

2^n Algorithms with exponential running time are not suitable for practical use. Usually these algorithms arise as brute force solution to the problems. Most often several innovative heuristics are to be applied to such algorithms so that the average case complexity of the algorithms become much less than exponential.

Figure 1.3 indicates the relative sizes of some of these functions.

n	logn	√n	nlogn	nlog²n	n²
10	3	3	30	90	100
100	6	10	600	3600	10,000
1000	9	31	9,000	81,000	1,000,000
10,000	13	100	130,000	1,690,000	100,000,000
100,000	16	316	1,600,000	25,600,000	ten billion
1,000,000	19	1,000	19,000,000	361,000,000	one trillion

Figure 1.3

Often the results of a mathematical analysis are not exact but approximate. The result may be an expression having a sequence of decreasing terms. In such a case, the largest term (which is also called the leading term) is the most important one. In fact, O-notation is developed precisely for such experience so that by using this notation concise statements can be made.

Suppose that an algorithm consists of three blocks. The first block is for initialization and takes a constant amount of time, say "c". The next block is a simple iteration whose time complexity is c.n. The last block has a time complexity of c(nlogn). The overall time complexity of the algorithm is

$$c + cn + cn\log n$$

$$\text{i.e., } cn\log n + O(n) = O(n\log n)$$

Note that, as far as analysis of algorithms is concerned, the exact value of "c" is not important because the analysis attempts to reach an approximate result only. Whenever f(n) is asymptotically large compared to another function g(n), f(n) + O(g(n)) typically means "approximately f(n)". The following observation can be applied to an algorithm consisting of several blocks where time complexities of these blocks are O(n), $O(n^2)$,.., $O(n^m)$ respectively.

Observation

If $A(n) = a_m n^m + \ldots + a_1 n + a_0$ is a polynomial of degree m then $A(n) = O(n^m)$.

Proof

The proof is based on a simple inequality

$$\begin{aligned} |A(n)| &\le |a_m| n^m + \ldots + |a_1| n + |a_0| \\ &\le (|a_m| + |a_{m-1}|/n + \ldots + |a_0|/n^m)\, n^m \\ &\le (|a_m| + \ldots + |a_0|)\, n^m \text{ for } n \geq 1 \end{aligned}$$

If c is made equal to $|a_m| + \ldots + |a_0|$ and $n_0 = 1$ then, $|A(n)| \le c|n^m|$ for all $n \ge n_0$.

2.3. Matrix as a 2-Dimensional Array

Mathematically, a matrix, "A", is a collection of elements "a_{ij}" for all i, j's such that $0 \le i < m$ and $0 \le j < n$. The matrix "A" is said to be of order m x n. Pictorially a matrix can be described as shown in Figure 2.5. The matrix "A" is also said to consist of "m" number of rows and "n" columns. The element a_{ij} is the element positioned at the i-th row and the j-th column in Figure 2.2. The elements a_{ii} for all "i" are called diagonal elements of "A". If m = n (i.e. number of rows and number of columns are equal) then, the matrix "A" is called a square matrix. Each row or column of a matrix is also known as its row vector or column vector respectively. Primitive matrix operations are (i) addition of two matrices, (ii) multiplication of two matrices, (iii) transposition of given matrix, (iv) evaluation of the determinant of a square matrix etc.

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a \ldots\ldots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & a \ldots\ldots & a_{1,n-1} \\ a_{2,0} & a_{2,1} & a \ldots\ldots & a_{2,n-1} \\ .. & & & \\ .. & & & \\ a_{m-2,0} & a_{m-2,1} & a \ldots\ldots & a_{m-2,n-1} \\ a_{m-1,0} & a_{m-1,1} & a \ldots\ldots & a_{m-1,n-1} \end{pmatrix}$$

Figure 2.5. Pictorial representation of a mxn matrix.

A matrix can be conveniently represented by a two-dimensional array. Moreover, the arithmetic computations involving matrices can be easily translated into algorithms using two-dimensional arrays. In the following, "C" functions are discussed to compute the sum and product of two matrices and to compute the transpose of a square matrix.

Let A & B be two 3 X 4 matrices as given in the following:

$$A = \begin{vmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{vmatrix} \qquad B = \begin{vmatrix} 9 & 1 & 0 & -5 \\ 6 & 2 & 10 & 0 \\ 9 & -15 & 5 & 9 \end{vmatrix}$$

$$C = A + B = \begin{vmatrix} 1+9 & 2+1 & 3+0 & 4+(-5) \\ 5+6 & 6+2 & 7+10 & 8+0 \\ 9+9 & 10+(-15) & 11+5 & 12+9 \end{vmatrix}$$

Note that $c_{i,j} = a_{i,j} + b_{i,j}$ for all i and j such that $0 \leq i < 3$ and $0 \leq j < 4$.
The following function "MAdd" adds two matrices "A" and "B" to produce a matrix "C". The order of all the matrices "A", "B" and "C" is m x n.

```
void MAdd (int A[10][10], int B[10][10], int C[10][10], int m, int n)
{
        int i,j;
        for(i = 0; i<m; i++)
                for (j = 0; j < n; j++)
                        C[i][j] = A[i][j] + B[i][j];
}
```

The product of two matrices "A" and "B" is defined only when the number of columns of "A" is equal to the number of rows in "B". In that case, the (i, j)-th element of the product "C", $c_{i,j}$ is the dot product of the i-th row vector of "A" and the j-th column vector of "B".

Dot product of a vector $X = (x_1, x_2, \ldots, x_m)$ and $Y = (y_1, y_2, \ldots, y_m)$ is defined as

$$X.Y = (x_1y_1 + x_2y_2 + x_3y_3 + \ldots + x_my_m)$$

If C = AXB then,

$$c_{i,j} = \Sigma_{k=0,n-1}\, a_{i,k} \cdot b_{k,j}$$ where "n" is the number of columns of "A".

Let A and B be 3 X 4 and 4 X 3 matrices respectively. Suppose A and B are given as follows:

$$A = \begin{vmatrix} 9 & 1 & 0 & -5 \\ 6 & 2 & 10 & 0 \\ 9 & -15 & 5 & 9 \end{vmatrix} \qquad B = \begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{vmatrix}$$

Then, "C" becomes

$$\begin{vmatrix} -37 & -32 & -27 \\ 84 & 102 & 120 \\ 74 & 82 & 90 \end{vmatrix}$$

Note that

$$= 9*1 + 1*4 + 0*7 + (-5)*10$$
$$= 9 + 4 + 0 - 50 = -37$$

Let "A" be a mxn matrix and "B" be a nxp matrix. From the earlier discussions, it follows that for a given i, j, C[i][j] may be computed by evaluating the sum of a series of k terms where each term is A[i][k]*B[k][j]. Clearly, "k" varies from 0 to (n-1). C[i][j] itself, has to be computed for all i, j, $0 \leq i < m$ and $0 \leq j < p$. The "C" function for computing

the product of two matrices is given in the following. The arrays "A", "B", "C" storing the matrices A, B and C are to be passed as parameters. In addition, three integers m, n, p are also to be provided so that the order of the matrices A and B are known to the function. The function assumes that the matrices in "A" and "B" are order m x n and n x p respectively.

```
void matrixMult( int A[10][10], int B[10][10], int C[10][10], int m, int n, int p)
{
        int i, j, k;
        for (i = 0; i < m; i++)
                for (j = 0; j < p; j++)
                {
                        C[i][j] = 0;
                        for (k = 0; k < n; k++)
                                C[i][j] += A[i][k]*B[k][j];
                }
}
```

The complexity of the algorithm "matrixMult" is O(mnp) because of three nested loops.

Next, consider the computation of the transpose of a square matrix "A". Transpose of an n X n square matrix A is a matrix B such that an element $b_{i,j}$ of B is equal to the element $a_{j,i}$ of A for $0 \leq i,j < n$. For example, if A is given as in the following:

$$A = \begin{vmatrix} 9 & 1 & 0 \\ 6 & 2 & 10 \\ 9 & -15 & 5 \end{vmatrix}$$

Then B, the transpose of A is given by

$$B = \begin{vmatrix} 9 & 6 & 9 \\ 1 & 2 & -15 \\ 0 & 10 & 5 \end{vmatrix}$$

So, a straightforward function to compute the transpose of a square matrix may be written as follows :

```
void transposeMatrix (int A[10][10], int B[10][10], int n)
{
        int i, j;
        for (i=0; i<n; i++)
                for (j=0; j<n; j++)
```

2.4.3 Large integer arithmetic

In some specialized applications, it may be required to manipulate very very large integers which can not be stored in variables of type "int" or "long" in "C" programming language. For example, suppose that 64 digit numbers are to be represented. Value of such integers is larger than the maximum number which can be stored in a variable of type "long". So, a different representation scheme is required for large integers. Note that a number can always be considered to be a special type of polynomial. For instance, a number such as 1024 can be denoted by the following expression.

$$1x10^3 + 0x10^2 + 2x10^1 + 4x10^0$$

Clearly, the expression is the value of $P(x) = x^3 + 2x + 4$ when $x = 10$. In fact, any decimal integer of "n" digits can be thought of as the value of a (n-1)degree polynomial $P(x) = \Sigma_{i=0,n-1} a_i x^i$ for $x = 10$ where each a_i represents a digit of the number. As a_i denotes a digit, $0 \leq a_i \leq 9$. Thus, a number, N, can be represented as a polynomial in which each term in the polynomial represents a non-zero digit and its position from the right end of the number. So, the following 30-digit number

410596400003007000000008020010

can be represented as shown in the following.

1,1	2,4	8,6	7,15	3,18	4,23	6,24	9,25	5,26	1,28	4,29

Addition of two large integers represented in this manner is very similar to addition of two polynomials. The only difference is that although the "coefficient" part of the "digits" is between 0 and 9 for the integers to be added, after being added the coefficient part of a digit may become more than 9. In such a case, the new digit is "olddigit % 10" and another digit ("olddigit/10, oldexponent + 1") is to be imported for addition. The details of addition procedure is left as an exercise.

2.5. Sparse Matrix Representation

A sparse matrix is one where most of its elements are zero. For example, the following 6X6 matrix A is a sparse matrix.

$$A = \begin{pmatrix} 0 & 0 & 9 & 0 & 0 & 0 \\ 5 & 0 & 0 & 1 & 0 & 8 \\ 0 & 2 & 0 & 0 & 4 & 0 \\ 0 & 1 & 19 & 0 & 0 & 0 \\ 1 & 5 & 0 & 0 & 6 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \end{pmatrix}$$

Such a matrix may be represented more economically (in terms of space) if conventional two-dimensional array representation of matrices is not used. The idea is, it is possible to store informations regarding non-zero elements only. If this is done, then the matrix may be thought of as an ordered list of non-zero elements only. Information about a non-zero element has three parts : (i) an integer representing its row, (ii) an integer representing its column and (iii) the data associated with this element. Such a 3-tuple can be implemented by structures in "C" in the following manner:

```
typedef struct element
{
        int row, col, val;
} element;
```

Then, a sparse matrix may be defined as

```
typedef struct spmat
{
        int no_of_elements;
        int no_of_rows, no_of_columns;
        element_data[100];
}spmat;
```

Thus, the 6X6 matrix, "A" may be described as a one-dimensional array, "a", such that a.no_of_elements is 12, a.no_of_columns is 6, a.no_of_rows is 6 and the array a.data may be depicted as shown in Figure 2.7.

0,2,9	1,0,5	1,3,1	1,5,8	2,1,2	2,4,4	3,1,1	3,2,19	4,0,1	4,1,5	4,4,6	5,2,-1

Figure 2.7. Representing sparse matrix using array of structures.

Since each element in the array is a 3-tuple, the contents of the array is pictorially shown by means of three integers separated by commas. To find out the saving in space that could be achieved by such a representation, consider the space required for storing a mXn integer matrix in a two dimensional array. Naturally, the size required for the array is m*n*(size of an integer) number of bytes. In the representation using one dimensional array, an array of size "p" is required where p = number of non-zero elements in the matrix. Each element in "p" needs room to store 3 integers. Hence, the total space required is 3p*(size of an integer). Clearly, the second representation becomes advantageous if

3p*(size of an integer) < mn*(size of an integer)

i.e. p < mn/3

In other words, if number of non-zero elements is less than one third of the total number of elements in the matrix then the "array of structure" representation is better in terms of space.

Although the non-zero elements may be stored in the array "data" in any order, keeping them ordered in some fashion may be advantageous for further processing. Note that in the example, a.data is arranged in increasing order of the row number of non-zero elements. Moreover, for elements with same row number, the array is arranged in order of increasing column number. If a function is to be written to compute total number of elements in each row of the matrix, then such a function would traverse the array only once to get the result.

However, if someone wants to know the number of elements in each column, a straight forward function would take $O(mn^2)$ time to find that out. As the array "data" is not ordered in increasing order of column number, the entire array is to be searched to find which elements belong to a given column number. The "C" function for finding number of elements in each column is given in the following.

```
void find(spmat *a)
{
        int j, i;
        int count = 0;
        for (i = 0; i < a->no_of_columns; i++)
        {
                count = 0;
                for (j = 0; j < a->no_of_elements; j++)
                        if (a->data[j].col == i)
                                count++;
                printf ("number of element in column %d is %d", i, count);
        }
}
```

Clearly, the number of comparisons is p*n (where "p" is number of non-zero elements in the matrix represented by "a") and $p = O(mn)$ in the worst case. Hence, the time complexity of the above function is $O(mn^2)$. However, a function for the same may be written for a mXn matrix represented conventionally using a two-dimensional matrix which will work in O(mn) time. It is natural, therefore, to ask whether the saving in space achieved by our representation is responsible for such poor performance in processing. A closer inspection, however, reveals that the complexity of the above function may be improved by using a programming trick. Let an array "col" be declared as int col[n] so that col[i] stores the number of elements in the i-th column. The algorithm is as follows. col[i] is initialized to 0 for all i, $0 \le i < n$.

If "a" is the sparse matrix declared as "spmat a" then, the elements of the array a.data is scanned one by one. If a.data[j].col is "i" for some "j" then col[i] is incremented. This step is repeated for all j. The following function uses this algorithm.

```
void colelements(spmat *a)
{
        int i, j;
        int col[100];
        for (i = 0; i < a->no_of_columns; i++)
                col[i] = 0;
        for (i = 0; i < a->no_of_elements; i++)
        {
                j = a->data[i].col;
                col[j] += 1;
        }
        for (i = 0; i < a->no_of_columns; i++)
                printf ("no. of elements in column %d is %d", i, col[i]);
}
```

The first and third "for-loop" make "n" iterations while the number of iterations in the second "for-loop" is "p" where "n" is the number of columns and "p" is the number of non-zero elements of the sparse matrix. The time complexity of the above function is O(n+n+p). In the worst case p = O(mn). Then, the complexity becomes O(n+n+O(mn)) = O(mn). Also, observe that this time-efficient function needs an additional space in the form of array "col".

The next problem to be dealt with is computing the transpose of a sparse matrix defined as "spmat *a". On the surface, this might seem to be an easy problem with a straight forward algorithm as follows.

```
void simpleTranspose (spmat *a, spmat *b)
{
        int i;
        b->no_of_rows = a->no_of_columns;
        b->no_of_columns = a->no_of_rows;
        b->no_of_elements = a->no_of_elements;
                for (i = 0; i < a->no_of_elements; i++)
                {
```



```
                        b-> data[i].row = a-> data[i].col;
                        b-> data[i].col = a->data[i].row;
                        b-> data[i]. val = a->data[i].val;
                }
}
```

If this function is executed on the matrix in Figure 2.7, then the transposed matrix becomes:

2,0,9	0,1,5	3,1,1	5,1,8	1,2,2	4,2,4	1,3,1	2,3,19	0,4,1	1,4,5	4,4,6	2,5,-1

Figure 2.8. The transpose of the matrix of Figure 2.7.

Although the algorithm "simpleTranspose" runs in O(mn) time in the worst case and the transpose is properly computed, the matrix pointed to by "b" suffers from a serious drawback. The array "b->data" is not arranged in the way "a->data" was. Therefore, the function "simpleTranspose" turns out to be unsatisfactory. The next function overcomes this problem by finding out elements in the original matrix for columns starting from 0 to n-1.

```
void transpose (spmat *a, spmat *b)
{
        int i, j, k = 0;
        b->no_of_rows = a->no_of_columns;
        b->no_of_columns = a->no_of_rows;
        b->no_of_elements = a->no_of_elements;
        for (i = 0; i < a->no_of_columns; i++)
                for (j = 0; j < a->no_of_elements; j++)
                {
                        if (a->data[j].col== i)
                        {
                                b->data[k].row = i;
                                b->data[k].col=a->data[j].row;
                                b->data[k].val = a->data[j].val;
                                k++;
                        }
                }
}
```

For an mxn sparse matrix with "p" non-zero elements, the function "transpose" runs in O(mnp) time. Observe that, the major problem in this function is that for a given "j", the index in b-> data where a->data[j] is to be stored is not known. Also, note that the columns in "a" becomes rows in "b". Let the number of elements with row number as "i" in "b" be denoted by r_i. Then, r_i is equal to the number of elements with column number as "i" in "a". The index in b->data from where elements of the k-th row in "b" are to be stored is $\sum_{i=0}^{k-1} r_i$.

That is, b->data[0] contains information about the elements whose row number is 0 (Assume there are some non-zero elements in 0th row). b->data[0+number of elements in 0th row] contains the first non-zero element of the next row. In other words, b->data contains the first non-zero element of the first row. Similarly b->[r_0+r_1] contains the first non-zero element in the 2nd row and so on. r in "b" is nothing but col[0] as computed in the function "colelements". Then,

$r_2 = r_1 + col[1]$

$r_3 = r_2 + col[2]$

Once r_i is computed, this information can be used to store the elements of a->data to corresponding elements b-> data.

```
void fastTranspose (spmat *a, spmat *b)
{
        int i, j;
        int S[100],col[100];
        for (i = 0; i < a->no_of_columns; i++)
                col[i] = 0;
        for (i = 0; i < a->no_of_elements; i++)
                col[a->data[i].col] += 1;
        S[0] = 0;
        for (i = 1; i < a->no_of_columns; i++)
                S[i] = S[i-1] + col[i-1];
        for (i = 0; i < a->no_of_elements; i++)
        {
                j = a->data[i].col;
                b->data[S[j]].row = a->data[i].col;
                b->data[S[j]].col = a->data[i].row;
                b->data[S[j]].val = a->data[i].val;
                S[j] += 1;
        }
        b-> no_of_rows = a->no_of_columns;
        b-> no_of_columns = a->no_of_rows ;
        b-> no_of_elements = a->no_of_elements;
}
```


2.6. Representation of a multi-dimensional array in a one-dimensional array.

Although programmers writing in a high level language (like C, Pascal, FORTRAN etc.) certainly use multidimensional arrays, computer memory (from where such arrays would be allocated) is conceptually only one-dimensional. That is, physically, computer memory can only be viewed as consecutive units of memory. In high level languages like "C", abstraction is provided so that a programmer can view data objects as a multidimensional array. But, a programmer writing programs in assembly level language can not define or use multi-dimensional arrays. In assembly language, usually memory can be reserved for a chunk of bytes or words or of such known units. Thus, even for programmers using high level languages, it is important to know how multi-dimensional arrays may be represented using one dimensional arrays,. Consider the following "C" definition for a five-dimensional array of integers.

```
int a[10][5][12][9][7];
```

Clearly, the maximum number of integers that can be stores in the array is

$10*5*12*9*7 = 37800$

Moreover, five indices are required to access any individual element of the array. If these indices are denoted by i_1, i_2, i_3, i_4 and i_5 then according to the definition of "C" language, the following inequalities must hold.

$0 \le i_1 < 10$,

$0 \le i_2 < 5$,

$0 \le i_3 < 12$,

$0 \le i_4 < 9$ and

$0 \le i_5 < 7$

In other words if the indices i_1, i_2, i_3, i_4, i_5 are valid for the defined array then, the 5-tuple (i_1, i_2, i_3, i_4, i_5) must belong to the following set:

[0..9] X [0..4] X [0..11] X [0..8] X [0..6]

Where [i..j] represents the set {k | k is an integer and $i \le k \le j$ }.

As has been discussed, a general "m" dimensional array may be defined as item a[l_1..u_1][l_2..u_2]....[l_m..u_m] where, for every j, the index i_j satisfies the following criterion: