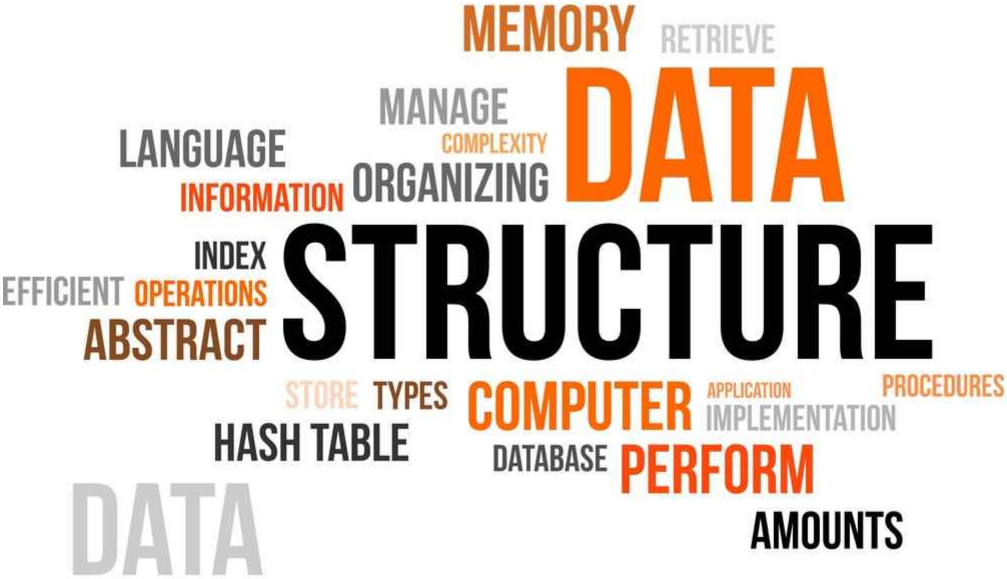


Data Structures

ALGORITHMS

Algorithms

INTRODUCTION

Data structure is the structural representation of logical relationships between elements of data. In other words a data structure Is a way of organizing data Items by considering its relationship to each other.

Data structure mainly specifies the structured organization of data, by providing accessing methods with correct degree of association. Data structure affects the design of both the structural and functional aspects of a program.

Algorithm + Data Structure = Program

ALGORITHM

Algorithm is a step-by-step finite sequence of instruction, to solve a well-defined computational problem.

That is, in practice to solve any complex real life problems; first we have to define the problems. Second step is to design the algorithm to solve that problem.

Writing and executing programs and then optimizing them may be effective for small programs. Optimization of a program is directly concerned with algorithm design. But for a large program, each part of the program must be well organized before writing the program. There are few steps of refinement involved when a problem is converted to program; this method is called stepwise refinement method. There are two approaches for algorithm design; they are *top-down and bottom-up algorithm* design.

TOP-DOWN ALGORITHM DESIGN

The principles of top-down design dictates that a program should be divided Into a main module and Its related modules. Each module should also be divided Into sub modules according to software engineering and programming style. The division of modules processes until the module consists only of elementary process that are Intrinsically understood and cannot be further subdivided.

In C, the Idea of top-down design Is done using functions. A C program Is made of one or more functions, one and only one of which must be named main. The execution of the program always starts and ends with main, but It can call other functions to do special tasks.

BOTTOM-UP ALGORITHM DESIGN

Bottom-up algorithm design Is the opposite of top-down design. It refers to a style of programming where an application Is constructed starting with existing primitives of the programming language, and constructing gradually more and more complicated features, until the all of the application has been written. That Is, starting the design with specific modules and build them Into more complex structures, ending at the top.

The bottom-up method Is widely used for testing, because each of the lowest-level functions Is written and tested first. This testing Is done by special test functions that call the low-level functions, providing them with different parameters and examining the results for correctness. Once lowest-level functions have been tested and verified to be correct, the next level of functions may be tested. Since the lowest-level functions already have been tested, any detected errors are probably due to the higher-level functions. This process continues, moving up the levels, until finally the main function Is tested.

ANALYSIS OF ALGORITHM

After designing an algorithm, it has to be checked and its correctness needs to be predicted; this is done by analyzing the algorithm. The algorithm can be analyzed by tracing all step-by-step instructions, reading the algorithm for logical correctness, and testing it on some data using mathematical techniques to prove it correct. Another type of analysis is to analyze the simplicity of the algorithm. That is, design the algorithm in a simple way so that it becomes easier to be implemented. However, the simplest and most straightforward way of solving a problem may not be sometimes the best one. Moreover there may be more than one algorithm to solve a problem. The choice of a particular algorithm depends on following performance analysis and measurements :

- Space complexity
- Time complexity

SPACE COMPLEXITY

Analysis of space complexity of an algorithm or program is the amount of memory it needs to run to completion.

Some of the reasons for studying space complexity are:

- If the program is to run on multi user system, it may be required to specify the amount of memory to be allocated to the program.
- We may be interested to know in advance that whether sufficient memory is available to run the program.
- There may be several possible solutions with different space requirements.
- Can be used to estimate the size of the largest problem that a program can solve.

The space needed by a program consists of following components.

- Instruction space : Space needed to store the executable version of the program and it is fixed.
- Data space : Space needed to store all constants, variable values and has further two components :
  - Space needed by constants and simple variables. This space is fixed.
  - Space needed by fixed sized structural variables, such as arrays and structures.

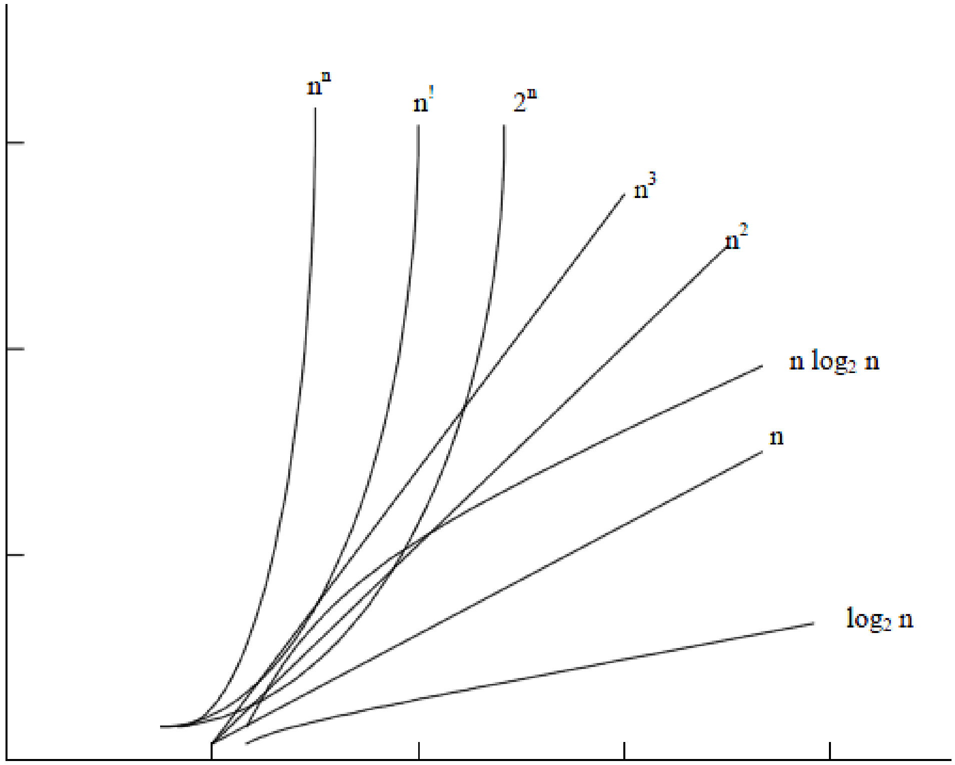
- Dynamically allocated space. This space usually varies.
- Environment stack space: This space is needed to store the information to resume the suspended (partially completed) functions. Each time a function is invoked the following data is saved on the environment stack :
  - Return address : i.e., from where it has to resume after completion of the called function.
  - Values of all lead variables and the values of formal parameters in the function being invoked .

The amount of space needed by recursive function is called the recursion stack space. For each recursive function, this space depends on the space needed by the local variables and the formal parameter. In addition, this space depends on the maximum depth of the recursion i.e., maximum number of nested recursive calls.

TIME COMPLEXITY

The time complexity of an algorithm or a program is the amount of time it needs to run to completion. The exact time will depend on the implementation of the algorithm, programming language, optimizing the capabilities of the compiler used, the CPU speed, other hardware characteristics/specifications and so on. To measure the time complexity accurately, we have to count all sorts of operations performed in an algorithm. If we know the time for each one of the primitive operations performed in a given computer, we can easily compute the time taken by an algorithm to complete its execution. This time will vary from machine to machine. By analyzing an algorithm, it is hard to come out with an exact time required. To find out exact time complexity, we need to know the exact instructions executed by the hardware and the time required for the instruction. The time complexity also depends on the amount of data inputted to an algorithm. But we can calculate the order of magnitude for the time required.

That is, our intention is to estimate the execution time of an algorithm irrespective of the computer machine on which it will be used. Here, the more sophisticated method is to identify the key operations and count such operations performed till the program completes its execution. A key operation in our algorithm is an operation that takes maximum time among all possible operations in the algorithm. Such an abstract, theoretical approach is not only useful for discussing and comparing algorithms, but also it is useful to improve solutions to practical problems. The time complexity can now be expressed as function of number of key operations performed. Before we go ahead with our discussions, it is important to understand the rate growth analysis of an algorithm, as shown in following figure.



The function that involves ‘n’ as an exponent, i.e., 2^n, n^n, n! are called exponential functions, which is too slow except for small size input function where growth is less than or equal to nc,(where ‘c’ is a constant) i.e.; n^3, n^2, n $\log_2$n, n, $\log_2$n are said to be polynomial.

Algorithms with polynomial time can solve reasonable sized problems if the constant in the exponent is small.

When we analyze an algorithm it depends on the input data, there are three cases :

- Best case
- Average case
- Worst case

In the best case, the amount of time a program might be expected to take on best possible input data.

In the average case, the amount of time a program might be expected to take on typical (or average) input data.

In the worst case, the amount of time a program would take on the worst possible input configuration.

Time Complexity of Bubble Sort

In the first pass of bubble sort, in worst case it must make N - l comparisons. In the next pass it must make N - 2 comparisons and so on. The algorithm terminates after maximum N- 1 pass (or earlier if there are no exchanges in a pass). Thus the total number of comparisons will be

$$(N-1)+(N-2)+(N-3)+.......3+2+1$$

$$= \frac{N(N-1)}{2} = \frac{N^2}{2} - \frac{N}{2} = O(N^2)$$

TIME-SPACE TRADE OFF

There may be more than one approach (or algorithm) to solve a problem. The best algorithm (or program) to solve a given problem is one that requires less space in memory and takes less time to complete its execution. But in practice, it is not always possible to achieve both of these objectives. One algorithm may require more space but less time to complete its execution while the other algorithm requires less time space but takes more time to complete its execution. Thus, we may have to sacrifice one at the cost of the other. If the space is our constraint, then we have to choose a program that requires less space at the cost of more execution time. On the other hand, if time is our constraint such as in real time system, we have to choose a program that takes less time to complete its execution at the cost of more space.

ASYMPTOTIC NOTATION

Asymptotic notation is the most simple and easiest way of describing the running time of an algorithm. It represents the efficiency and performance of an algorithm in a systematic and meaningful manner. Asymptotic notations describe time complexity in terms of three common measures, best case (or 'fastest possible'), worst case (or 'slowest possible'), and average case (or 'average time'). The three most important asymptotic notations are:

- Big-Oh notation
- Omega notation
- Theta notation

Big “O” Notation

Big O is a characteristic scheme that measures properties of algorithm complexity performance and/or memory requirements. The algorithm complexity can be determined by eliminating constant factors in the analysis of the algorithm. Clearly, the complexity function f(n) of an algorithm increases as ‘n’ increases.

Let us find out the algorithm complexity by analyzing the sequential searching algorithm. In the sequential search algorithm we simply try to match the target value against each value in the memory. This process will continue until we find a match or finish scanning the whole elements in the array. If the array contains ‘n’ elements, the maximum possible number of comparisons with the target value will be ‘n’ i.e., the worst case. That is the target value will be found at the nth position of the array.

$$f(n) = n$$

i.e., the worst case is when an algorithm requires a maximum number of iterations or steps to search and find out the target value in the array.

The best case is when the number of steps is less as possible. If the target value is found in a sequential search array of the first position (i.e., we need to compare the target value with only one element from the array) - we have found the element by executing only one iteration (or by least possible statements)

$$f(n) = 1$$

Average case falls between these two extremes (i.e., best and worst). If the target value is found at the n/2nd position, on an average we need to compare the target value with only half of the elements in the array, so

$$f(n) = n/2$$

The complexity function f(n) of an algorithm increases as 'n' increases. The function f(n)= O(n) can be read as "f of n is big O of n" or as "f(n) is of the order of n". The total running time (or time complexity) includes the initializations and several other iterative statements through the loop.

Then, $f(n) = O(n^k)$

Based on the time complexity representation of the big Oh notation, the algorithm can be categorized as :

1. Constant time O(1)
2. Logarithmic time Olog(n)
3. Linear time O(n)
4. Polynomial time $O(n^c)$, if $c > 1$
5. Exponential time $O(c^n)$, if $c > 1$

Limitation Of Big "O" Notation

Big O Notation has following two basic limitations :

- It contains no effort to improve the programming methodology. Big O Notation does not discuss the way and means to improve the efficiency of the program, but it helps to analyze and calculate the efficiency (by finding time complexity) of the program.
- It does not exhibit the potential of the constants. For example, one algorithm is taking $1000n^2$ time to execute and the other n^3 time. The first algorithm is $O(n^2)$, which implies that it will take less time than the other algorithm which is $O(n^3)$. However in actual execution the second algorithm will be faster for $n < 1000$.

We will analyze and design the problems in data structure. As we have discussed to develop a program of an algorithm, we should select an appropriate data structure for that algorithm.

Little O Notation

The little O is denoted as o. It is defined as : Let, f(n) and g(n) be the non negative functions then

$$n \xrightarrow{\lim} \infty \frac{f(n)}{g(n)} = 0$$

such that f(n) = o(g(n))

Theta notation

$$f(n) = \theta(g(n))$$

read as f of n equal lo theta of g(n) if and only if, there exists positive constants Q.C_1 and C_2 such that for all n > n_0

$$C_1 \mid g(n) \mid \leq f(n) \leq C_2 \mid g(n) \mid$$

If f(n) = θ(g(n)) and g(n) both an upper and lower bound on f(n). i.e. worst and best cases require the same amount of time to within constant factor.

Example

Show that $n\log(n) \in (\log(n!))$

Solution

θ notation is used for tight bound

We know $n! < n^n$

i.e. $\log(n!) < \log(n^n) < n\log n$

log(n!)=log(1) + log (2) + ... log(n – 1) + log(n)

We can get upper bound by

log(1) + log (2) + ... log(n) ≤ log(n) + log(n) + ... log(n) ≤ nlog(n)

We can get lower bound by throwing away first half of the sum

log(1) + log(2) + ... log(n/2) + ...log(n) ≥ log(n/2)

+.....log(n) ≥ log(n/2) + ... + log(n/2) = (n/2)*log(n/2)

For the same values of n there is a tight lower bound and also for same values of n there is tight upper bound. hence $n\log n \in \theta(\log(n!))$

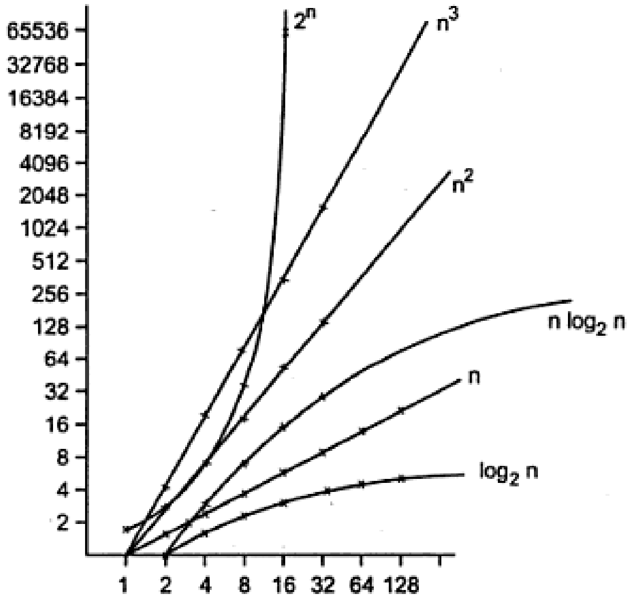
EFFICIENCY OF ALGORITHM

If we have two algorithms that perform same task, and the first one has a computing time of O(n) and the second of O(n^2), then we will usually prefer the first one.

The reason for this is that as n increases the time required for the execution of second algorithm will get far more than the time required for the execution of first.

We will study various values for computing function for the constant values. The graph given below will indicate the rate of growth of common computing time functions.

n	$\log_2 n$	n logn	n^2	n^3	2^n
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65536
32	5	160	1024	32768	2147483648



Notice how the times O(n) and O(n $\log_2$n) grow much more slowly than the others. For large data sets algorithms with a complexity greater than O(n $\log_2$n) are often impractical. The very slow algorithm will be the one having time complexity 2^n.

Example

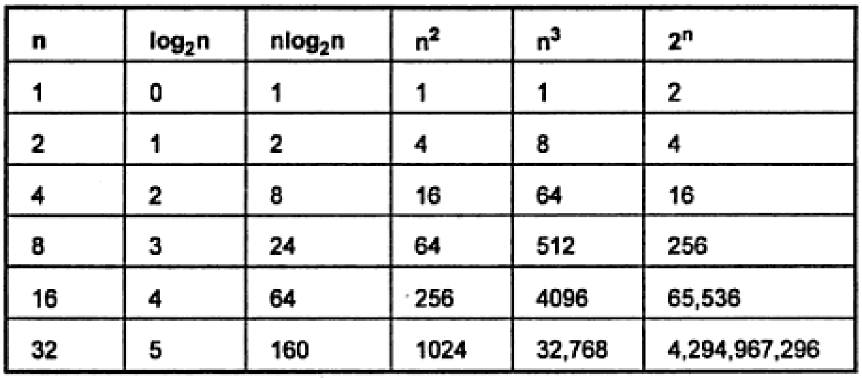
$$f(n) = 3n^3 + 2n^2 + 4n + 3$$
$$= 3n^3 + 2n^2 + O(n), \text{ as } 4n + 3 \text{ is of } O(n)$$
$$= 3n^3 + O(n^2), \text{ as } 2n^2 + O(n) \text{ is } O(n^2)$$
$$= O(n^3)$$

Example

Which time complexity shows poor performance? Why?

Solution

See table.



n	$\log_2 n$	$n\log_2 n$	n^2	n^3	2^n
1	0	1	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65,536
32	5	160	1024	32,768	4,294,967,296

Thus as the value of n increases the value of 2^n becomes the largest among all the time complexity values. The program, which gives the time complexity value large, is supposed to be the slow one that is why it is said that the program having the time complexity 2^n is poor in performance.

Example

Find the frequency count for the following piece of code.

sum = 0;

for (i = 1; i<=n; i++)

sum = sum + a[i]

Solution

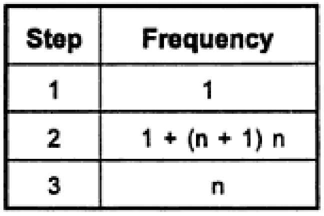
Let us write stepwise

Step1: sum = 0;

Step 2: for (i = 1; i<=n; i++)

Step 3: sum = sum + a[i]

Total frequency count = l + l + (n + l) + n + n = 3n + 3. The time complexity will be O(n).



Step	Frequency
1	1
2	1 + (n + 1) n
3	n

Example (AMIE S11, 4 marks)

Suppose $T_1(n)$ and T_2 (n) are the time complexities of two program fragments P_1 and P_2, where $T_1(n) = O(f(n))$ and $T_2 = O(g(n))$. What is the time complexity of program fragment P_1 followed by P_2?

Solution

The time complexity of the program fragment P_1 followed by P_2 is given by $T_1(n) + T_2(n)$.

$T_1(n) \leq C.f(n)$ for some positive integer C and positive and positive integer n_1, such that $n \geq n_1$.

$T_2(n) \leq D.g(n)$ for some positive integer d and positive and positive integer n_2, such that $n \geq n_2$.

Let $n_0 = \max(n_1, n_2)$. Then $T_1(n) + T_2(n) \leq C.f(n) + D.g(n)$ for $n > n_0$ i.e. $T_1(n) + T_2(n) \leq (C + D) \max(f(n), g(n))$ for $n>n_0$.

Hence $T_1(n) + T_2(n) = 0 (\max(f(n), g(n))$

Example (AMIE W10 6 marks)

Suppose $f_1(n) \in O(t(n))$ and $f_2(n) \in O(t(n))$. Which one of the following is true? In case an option is false, give reasons that why is it false.

(i) $f_1(n) + f_2(n) \in O(t(n))$

(ii) $f_1(n) - f_2(n) \in O(t(n))$

(iii) $f_1(n) / f_2(n) \in O(1)$

(iv) $f_1(n) \in O(f_2(n))$

Solution

(i) If $f_1(n)=Og_1(n)$ and $f_2(n) = Og_2(n)$ then $f_1(n) + f_2(n)= O(\max (g_1(n), g_2(n))$. here since both $f_1$and f_2 are O (t(n)), max (t(n), t(n)) = t(n). Thus

$$f_1(n) + f_2(n) \in O(t(n)) \text{ True}$$

(ii) Similarly

$$f_1(n) - f_2(n) \in Ot(n) \text{ True}$$

(iii) Here $f_1(n) / f_2(n) = const.$ since both are O(t(n))

Hence $f_1(n) / f_2(n) = O(1)$ True

(iv) Since $f_1(n) \in O(t(n))$ and $f_2(n) \in O(t(n))$

$$f_n(n) \in O(cf_2(n)) \text{ where c is a constant. True}$$

Example (AMIE W10, S11, 4 marks)

Suppose P(n) = $a_0 + a_1 n + a_2 n^m$*, that is, suppose degree P(n) = m. Prove that P(n) = O(*n^m*).*

Solution

Let $b_0 = |a_0|\, b_1 = |a_1| ......b_m = |a_m|$

Then for n≥1

$$P(n) \le b_0 + b_1 n + b_2 n^2 ....b_m n^m$$

$$= \left(\frac{b_0}{n^m} + \frac{b_1}{n^{m-1}} ....b_m \right) n^m \le (b_0 + b_1 + ...b_m) n^m = Mn^m$$

where $M = |a_0| + |a_1| + |a_2| ....|a_m|$

Hence $P(n) = O(n^m)$

SPACE COMPLEXITY

Another useful measure of an algorithm is the amount of storage space it needs. The space complexity of an algorithm can be computed by considering the data and their sizes. Again we are concerned with those data items which demand for maximum storage space. A similar notation 'O' is used to denote the space complexity of an algorithm. When computing for storage requirement we assume each data element needs one unit of storage space. While as the aggregate data items such as arrays will need n units of storage space n is the number of elements in an array. This assumption again is independent of the machines on which the algorithms are to be executed.

The space complexity of a computer program is the amount of memory required for its proper execution. The important concept behind space required is that unlike time, space can be reused during the execution of the program. As discussed, there is often a trade-off between the time and space required to run a program.

In formal definition, the space complexity is defined as follows:

Space complexity of a Turing Machine: The (worst case) maximum length of the tape required to process an input string of length n.

Example

Consider the following example: Binary Recursion (A binary-recursive routine (potentially) calls itself twice).

1. If n equals 0 or 1, then return 1
2. Recursively calculate f (n−1)
3. Recursively calculate f (n−2)
4. Return the sum of the results from steps 2 and 3.

Time Complexity: O(exp n)

Space Complexity: O(exp n)

SPARSE ARRAY

If we are reading or writing two-dimensional array, two loops are required. Similarly the array of ‘n’ dimensions would require ‘n’ loops. The structure of the two dimensional array is illustrated in the following figure :

int A[10][10];

A_{00}	A_{01}	A_{02}						A_{08}	A_{09}
A_{10}	A_{11}								A_{19}
A_{20}									
A_{30}									
									A_{69}
A_{70}								A_{78}	A_{79}
A_{80}	A_{81}						A_{87}	A_{88}	A_{89}
A_{90}	A_{91}	A_{92}				A_{96}	A_{97}	A_{98}	A_{99}

Two Dimensional Array

Sparse array is an important application of arrays. A sparse array is an array where nearly all of the elements have the same value (usually zero) and this value is a constant. One-dimensional sparse array is called sparse vectors and two-dimensional sparse arrays are called *sparse matrix*.

The main objective of using arrays is to minimize the memory space requirement and to improve the execution speed of a program. This can be achieved by allocating memory space for only non-zero elements.

For example a sparse array can be viewed as

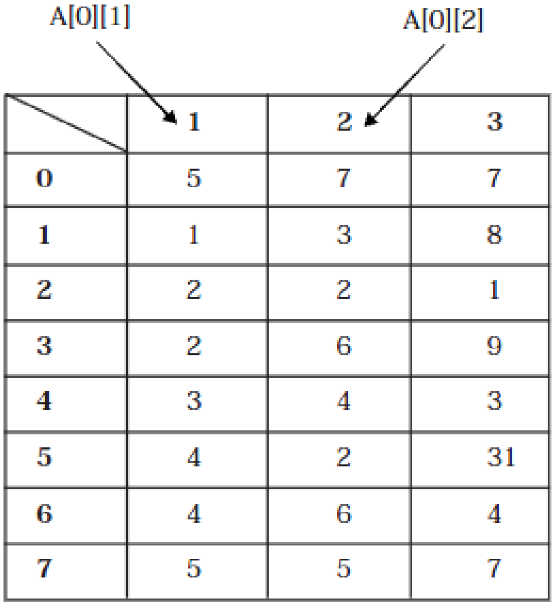
$$\begin{matrix} 0 & 0 & 8 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 9 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 & 0 \\ 0 & 31 & 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 7 & 0 & 0 \end{matrix}$$

We will store only non-zero elements in the above sparse matrix because storing all the elements of the sparse array will be consisting of memory sparse. The non-zero elements are stored in an array of the form.

A[0......n][1......3]

Where ‘n’ is the number of non-zero elements in the array. In the above Fig. of two dimensional array ‘n = 7’.

The space array given in two dimensional array above may be represented in the array A[0......7][1.....3].



	1	2	3
0	5	7	7
1	1	3	8
2	2	2	1
3	2	6	9
4	3	4	3
5	4	2	31
6	4	6	4
7	5	5	7

The element A[0][1] and A[0][2] contain the number of rows and columns of the sparse array. A[0][3] contains the total number of nonzero elements in the sparse array.

A[1][1] contains the number of the row where the first nonzero element is present in the sparse array. A[1][2] contains the number of the column of the corresponding nonzero element. A[1][3] contains the value of the nonzero element. In the Fig. of two dimensional array, the first nonzero element can be found at 1st row in 3rd column.

FILES AND RECORDS

A file is typically a large list that is stored in the external memory (e.g.. a magnetic disk) of a computer.

A record is a collection of information (or data items) about a particular entity. More specifically, a record is a collection of related data items, each of which is called a filed or attribute and a file is a collection of similar records.

Although a record is a collection of data items, it differs from a linear array in the following ways:

- A record may be a collection of non-homogeneous data: i.e.. the data items in a record may have different data types.
- The data items in a record are indexed by attribute names, so there may not be a natural ordering of its elements.

STRINGS

In computer terminology the term 'string' refers to a sequence of characters. A finite set of sequence (alphabets, digits or special characters) of zero or more characters is called a string. The number of characters in a string is called the length of the string. If the length of the string is zero then it is called the empty string or null string.

Strings are stored or represented in memory by using following three types of structures:

- Fixed length structures
- Variable length structures with fixed maximum
- Linear structures

Fixed Length Representation

In fixed length storage each line is viewed as a record, where all records have the same length. That is each record accommodates maximum of same number of characters.

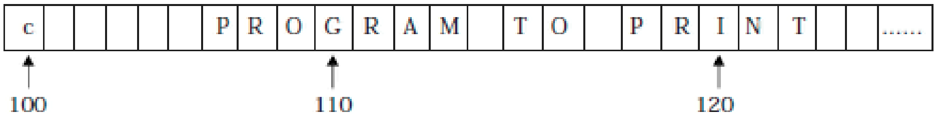
The main advantage of representing the string in the above way is :

- To access data from any given record easily.
- It is easy to update the data in any given record.

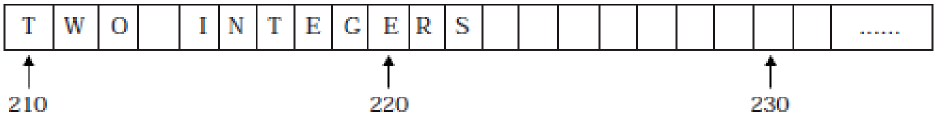
The main disadvantages are :

- Entire record will be read even if most of the storage consists of inessential blank space. Time is wasted in reading these blank spaces.
- The length of certain records will be more than the fixed length. That is certain records may require more memory space than available.

Fig. (b) is a representation of input data which is in Fig. (a) in a fixed length (records) storage media in a computer.



(a) Input Data



(b) Fixed Length Representation

Variable Length Representation

In variable length representation, strings are stored in a fixed length storage medium. This is done in two ways.

1. One can use a marker, (any special characters) such as two-dollar sign ($$), to signal the end of the string.
2. Listing the length of the string at the first place is another way of representing strings in this method.

c		p	r	o	g	r	a	m		t	o		P	r	i	n	t		t	w	o		i	n	t	e	g	e	r	s	$$

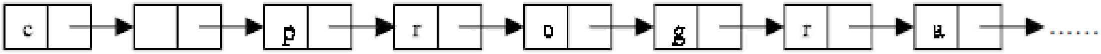
String representation using marker

31	c		p	r	o	g	r	a	m		t	o		p	r	i	n	t		t	w	o		i	n	t	e	g	e	r	s

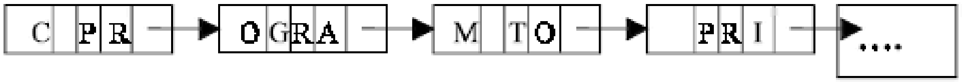
String representation by listing the length

Linked List Representations

In linked list representations each characters in a string are sequentially arranged in memory cells, called nodes, where each node contain an item and link, which points to the next node in the list (i.e., link contain the address of the next node).



One character per node



Four character per node

ASSIGNMENT

Q.1. (AMIE S12, 5 marks): Define Big O notation and what is its utility in analysis of algorithms?

Q.2. (AMIE S13, 4 marks): What is asymptotic little O notation (o)? What is big O notation?

Q.3. (AMIE S07, 12 marks): What do you mean by efficiency of an algorithm? How can you compare the efficiency of two algorithms? Explain the concept of best case, average case and worst case time complexity.

Q.4. (AMIE S12, 6 marks): What is worst case and average case analysis?

Q.5. (AMIE W07, 10 marks): Define (i) Time complexity (ii) space complexity (iii) array representation of strings (iv) record (v) abstract data type

Q.6. (AMIE S12, 5 marks): Describe briefly three types of structures used for storing strings.

Q.7. (AMIE S13, 6 marks): What is time complexity of an algorithm? Calculate run time complexity of bubble sort.

Q.8. (AMIE S11, 5 marks): Order the following function by growth rate: N, $\sqrt{N}$, $N^{1.5}$, N^2, NlogN, NloglogN, Nlog(N^2), 2/N, 2^N, 37

(For online support such as eBooks, video lectures, audio lectures, unsolved papers, quiz, test series and course updates, visit www.amiestudycircle.com)