Unit-1
Introduction to Data Structures & Algorithms:

1. Data Types:

The data type of a value or a variable is an attribute that tells what kind of data that value can have. Data types include the storage classifications like integers, floating point values, strings, characters etc. Each variable has an associated data type. Each data type requres different amounts of memory and has some specific operations which can be performed over it.

(a) Basic Data types → The data-types that are pre-defined which are immediately available to users by compiler while programming are basic data types. Some of the basic data types are as follows:-

i) Integer → It is a data type used to store integer values. If the floating point numbers are tried to store in this data type then it can not store floating numbers. It will terminate or escape numbers after decimal point and will only store fractional value.

ii) Character → It stores a single character and requires single byte of memory in almost all compilers.

iii) float → It is used to store decimal numbers with single ~~double~~ precision. It takes 4-byte memory space.

iv) Double → It is used to store decimal numbers with double precision. It takes 8-byte memory space

(b) User-defined data types → The data types that are defined by the user in the program are called derived data type or user-defined data type. Following are some of the user-defined data types:

i) Class → The user defined data type that holds data members and member functions is called class. A ~~co~~ class can be accessed and used by creating object of that class.

ii) Structure → A structure is a user defined data type in C/C++. A structure creates a data type that can be used to group items of possibly different types into a single type.

iii) Union → Union is a user-defined data type similar to structure. In union all members share the same memory location.

iv) Enumeration → Enumeration (or enum) is a user defined data type in C. It is mainly used to assign names to integral constants, the names make program easy to read and maintain.

2. Data Structure:

Data structure is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way. Data structure is about rendering data elements in terms of some relationship, for better organization and storage.

For example:- Let we have some data which has student's name "Ramesh" and age 24. Here "Ramesh" is of string data type and 24 is of integer data type. We can organize this data as a record like Student's record, which will have both name and age in it. Now we can collect and store Student's record in a database as a data structure.

In simple language, Data structures are structures programmed to store ordered data, so that various operations can be performed on it eaisly. It represents the knowledge of data to be organized in memory. It should be designed and implemented in such a way that it reduces the complexity and increases the efficiency.

Algorithm + Data Structure = Program

- Uses of data structure:
  - i) Data structure is used to reduce complexity of programs.
  - ii) It is used to increase efficiency of programs.
  - iii) Data structures are used as a framework for organizing and storing information in virtual memory forms.
  - iv) It prevents data collision by eliminating ambiguous data.
  - v) It is used to arrange the data in an efficient and effective manner.

3. Abstract data types (ADTs):

Def^n → Abstract data type (ADT) is a type or a class for objects whose behaviour is defined by a set of value and a set of operations.

It does not specify how data will be organized in the memory and what algorithms will be used for implementing the operations.

- Uses:
  - i) ADTs are used in the design and analysis of algorithms, data structures and software systems.
  - ii) It provides concept for data abstraction and data hiding.

- Advantages/Benefits of using ADTs:
  - i) Modularity
  - ii) Precise specifications.
  - iii) Information hiding.
  - iv) Simplicity
  - v) Integrity
  - vi) Implementation independence.

4. Concept of dynamic memory allocation:

The memory allocation that we do during compile time is static memory allocation. If the memory allocation is done during runtime is called dynamic memory allocation, In static memory allocation the memory used by the program is fixed i.e, we could not decrease or increase the size of memory during the execution of program. In many applications it is not possible to predict how much memory would be needed by the program at run time, now in this situation following two types of problems may occur.

i) If the number of values to be stored is less than the size of array then there will be the wastage of memory.

ii) If we want to store more values than the size of array we specified then we can't.

To overcome these problems we should be able to allocate memory at runtime. The process of allocating memory at the time of execution is called dynamic memory allocation. The allocation and release of this memory space can be done with the help of some built-in-functions whose prototypes are found in alloc.h and stdlib.h header files.

Pointers play an important role in dynamic memory allocation because we can access the dynamically allocated memory only through pointers.

⊛ Some built-in-functions used in dynamic memory allocation:

i) malloc() → This function is used to allocate memory dynamically.

<u>Syntax</u>: pointer_variable = (datatype*) malloc (specified_size);

ii) calloc () → The calloc() is used to allocate multiple blocks of memory. It is somewhat similar to malloc() except for two differences. The first one is that malloc() takes only one argument while calloc() takes two arguments. The first argument specifices number of blocks and the second one specifies the size of each block.

For example: ptr=(int *) calloc (5, sizeof(int));

The other difference between calloc() and malloc() is that the memory allocated by malloc() contains garbage value while the memory allocated by calloc() is initialized to zero.

iii) realloc() → The function realloc() is used to change the size of memory block. It alters the size of the memory block without losing the old data. This is known as reallocation of memory.

This function takes two arguments, first is a pointer to the block of memory that was previously allocated by malloc() or calloc() and second one is the new size for that block.

For example: ptr= (int *) realloc (ptr, newsize);

5. Introduction to Algorithms:

Definition → A process or set of rules to be followed in calculations or other problem-solving operations by a computer is called algorithm.

⊛. What is good algorithm?

Ans:- A algorith is called good if it contains following characteristics:

Input → An algorithm should have zero or more stated inputs.

Output → An algorithm should produce desired output.

Precision → The number of steps in an algorithm should be precisely defined (stated).

Uniqueness → The results of each step are uniquely defined and only depend on input.

Fitness → The algorithm should stop after executing finite no of steps.

Effective → Each step in the algorithm should be effective.

⊛. Different structures used in algorithms:

i) Search → Algorithm to search an item in a data structure.
ii) Sort → Algorithm to sort items in a certain order.
iii) Insert → Algorithm to insert item in a data structure.
iv) Update → Algorithm to update an existing item in a data structure.
v) Delete → Algorithm to delete an existing item from a data structure.

6. Asymptotic notations and common functions:

ⓐ. Algorithm Complexity → Suppose X is an algorithm and n is a size of input data, the time and space used by the algorithm X are the two main factors which decide the efficiency of X.

i) Time factor → Time is measured by counting the number of key operations such as comparisions in the sorting algorithm.

ii) Space factor → Space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm f(n) gives the running time and/or the storage space required by the algorithm in terms of n as the size of input data.

1) Space Complexity:

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components.

i). A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size etc.

ii) A variable part is a space required by variables, whose size depends on the size of the problem. For example; dynamic memory allocation, recursion stack space etc.

Space complexity $S(P)$ of any algorithm P is $S(P) = C + SP(I)$, where C is the fixed part and $S(I)$ is the variable part of the algorithm, which depends on instance characteristic I. Following is a simple example that tries to explain the concept—

```
Algorithm: SUM(A,B)
  Step1 — START
  Step2 — C ← A+B+10
  Step3 — Stop
```

Here, we have three variables A, B and C and one constant 10. Hence $S(P) = 1+3$. Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

#2) Time Complexity:

Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function $T(n)$, where $T(n)$ can be measured as the number of steps, provided each step consumes ~~cont~~ constant time.

For example, addition of two n-bit integers takes n steps. Consequently, the total computational time is

$T(n) = c*n$, where c is the time taken for the addition of two bits. Here, we observe that $T(n)$ grows linearly as the input size increases.

#3) Asympotatic analyses and asympotatic notations:

Asympotatic analysis; of an algorithm refers to defining the mathematical framing of it's run-time performance. Using asympotatic analyses, we can very well conclude the best case, average case and worst case scenario of an algorithm. Usually the time required by an algorithm falls under three types:-

i) Best case → Minimum time required for program execution.
ii) Average case → Average time required for program execution.
iii) Worst case → Maximum time required for program execution.

⊛ Asymptotic Notations:

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- Big Oh Notation, O
- Omega Notation, Ω
- Theta Notation, Θ

Big Oh Notation (denoted by O):

Big O notation is a mathematical notation that describes the limiting behaviour of a function when the argument tends towards a particular value or infinity.

In computer science, big O notation is used to classify algorithms according to how their running time or space ~~grows~~ requirement grows as the input size grows. In analytical number theory, big O notation is often used to express a bound on the difference between an arithmetical function and a better understood approximation.

When we have only asymptotic upper bound then we use O notation. If f and g are any two functions from set of integers to set of integers then function $f(x)$ is said to be big Oh of $g(x)$.

i.e., $f(x) = O(g(x))$ if and only if there exists two positive constants c and x_0 such that $x >= x_0, f(x) <= c * g(x)$.

This above relation shows that $g(x)$ is an upper bound of $f(x)$.

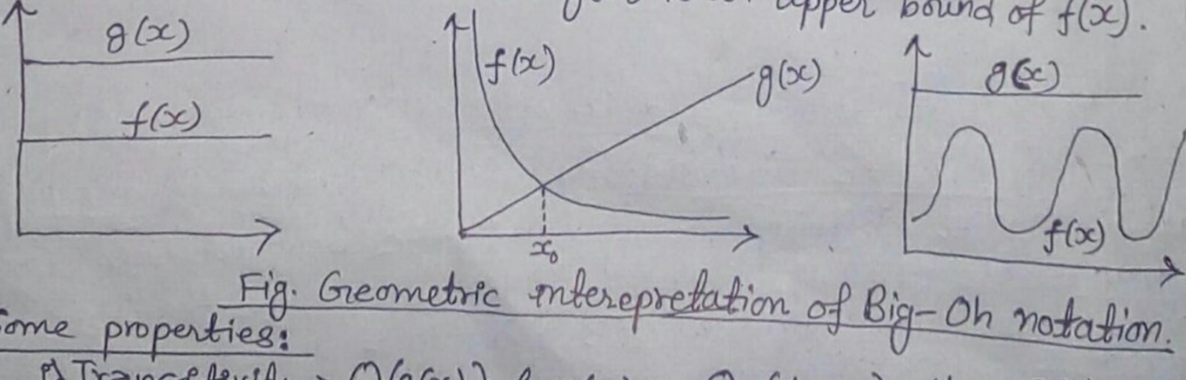

Fig. Geometric interepretation of Big-Oh notation.

Some properties:

i) Transitivity → $O(g(x))$ & $g(x) = O(h(x))$ then, $f(x) = O(h(x))$

ii) Reflexivity → $f(x) = O(f(x))$

iii) Transpose symmetry → $f(x) = O(g(x))$ if and only if $g(x) = \Omega(f(x))$.

⇒ $O(1)$ is used to denote constants.

Example: Find big oh of given function $f(n) = 3n^2 + 4n + 7$

Solution

we have, $f(n) = 3n^2 + 4n + 7 <= 3n^2 + 4n + 7 <= 14n^2$

$$f(n) <= 14n^2$$

where, $c = 14$ and $g(n) = n^2$, thus,

$$f(n) = O(g(n)) = O(n^2).$$