Unit I

Basic Terminologies & Introduction to Algorithm and Data Organisation

Unit I: Basic Terminologies & Introduction to Algorithm and Data Organisation (06 Hrs.)

Algorithm specification, Recursion, Performance analysis, Asymptotic Notation - The Big-O, Omega and Theta notation, Programming Style, Refinement of Coding - Time-Space Trade Off, Testing, Data Abstraction

1.1 Algorithm specification

The concept of an algorithm is fundamental to computer science. Algorithms exist for many common problems, and designing efficient algorithms plays a crucial role in developing large-scale computer systems. Therefore, before we proceed further we need to discuss this concept more fully. We begin with a definition.

Definition: An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:

(1) Input. There are zero or more quantities that are externally supplied.

(2) Output. At least one quantity is produced.

(3) Definiteness. Each instruction is clear and unambiguous.

(4) Finiteness. If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.

(5) Effectiveness. Every instruction must be basic enough to be carried out, in princi-ple, by a person using only pencil and paper. It is not enough that each operation be definite as in (3); it also must be feasible.

In computational theory, one distinguishes between an algorithm and a program, the latter of which does not have to satisfy the fourth condition. For example, we can think of an operating system that continues in a wait loop until more jobs are entered. Such a program does not terminate unless the system crashes. Since our programs will always terminate, we will use algorithm and program interchangeably in this text.

We can describe an algorithm in many ways. We can use a natural language like English, although, if we select this option, we must make sure that the resulting instructions are definite. Graphic representations called flowcharts are another possibility, but they work well only if the algorithm is small and simple. In this text we will present most of our algorithms in C, occasionally resorting to a combination of English and C for our specifications. An examples will help to illustrate the process of translating a problem into an algorithm.

Example 1.1 [Selection sort]: Suppose we must devise a program that sorts a set of n >= 1 integers. A simple solution is given by the following:

From those integers that are currently unsorted, find the smallest and place it next in the sorted list. Although this statement adequately describes the sorting problem, it is not an algorithm since it leaves several unanswered questions. For example, it does not tell us where and how the integers are initially stored, or where we should place the result. We assume that the integers are stored in an array, list, such that the ith integer is stored in the ith position, list [i], 0 <= i < n.
Program 1.1 is our first attempt at deriving a solution. Notice that it is written partially in C and partially in English.

```
for (i = 0; i < n; i++)
    {
        Examine list[i] to list[n-l] and suppose that the
        smallest integer is at list[min];
        Interchange list[i] and list[min];
    }
```

Program 1.1: Selection sort algorithm

1.2 Recursion

A recursive algorithm calls itself which usually passes the return value as a parameter to the algorithm again. This parameter is the input while the return value is the output.
Recursive algorithm is a method of simplification that divides the problem into sub-problems of the same nature. The result of one recursion is the input for the next recursion. The repletion is in the self-similar fashion. The algorithm calls itself with smaller input values and obtains the results by simply performing the operations on these smaller values.
Generation of factorial, Fibonacci number series are the examples of recursive algorithms.
Example: Writing factorial function using recursion

```
int factorial(int n)
{
return n * factorial(n-1);
}
```

Program 1.2: Factorial function

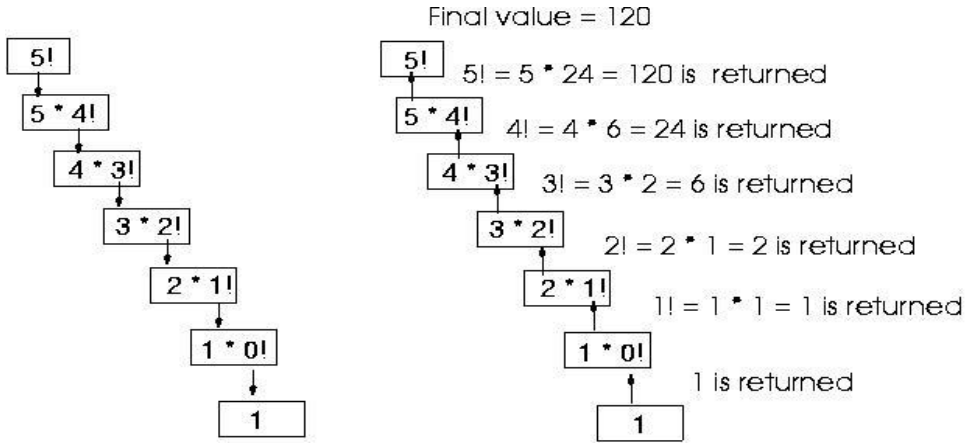

Figure 1.1

1.3 Performance analysis

Performance analysis of an algorithm is the process of calculating space and time required by that algorithm.It is measured in terms of **space complexity** and **time complexity.**

1.3.1 **Space Complexity:** The space complexity of a program is the amount of memory it needs to run to completion.

Generally, when a program is under execution it uses the computer memory for THREE reasons. They are as follows.

1. **Instruction Space:** It is the amount of memory used to store compiled version of instructions.
2. **Environmental Stack:** It is the amount of memory used to store information of partially executed functions at the time of function call.
3. **Data Space:** It is the amount of memory used to store all the variables and constants.

If any algorithm requires a fixed amount of space for all input values then that space complexity is said to be Constant Space Complexity.
If the amount of space required by an algorithm is increased with the increase of input value, then that space complexity is said to be Linear or Variable Space Complexity.

$$S(P) = C + S_P(I)$$

Where C is Fixed Space Requirements and $S_P(I)$ is Variable Space Requirements
Fixed Space Requirements (C) is Independent of the characteristics of the inputs and outputs and Variable Space Requirements ($S_P(I)$) depend on the instance characteristic I

Example: Program 1.3 is a recursive function for addition. Figure 1.2 shows the number of bytes required for one recursive call.

```
float rsum( float list[], int n)
        {
                if (n) return rsum (list, n-1) + list [n-1];
                return 0;
        }
```

Program 1.3: Recursive function for summing a list of numbers

Type	Name	Number of bytes
Parameter: float	list[]	2
Parameter: Integer	n	2
Return address: (used internally)		2(unless a far address)
		6

Figure 1.2: Space needed for one recursive call of program 1.3

1.3.2 **Time Complexity:** The time complexity of a program is the amount of computer time it needs to run to completion.

Generally, the running time of an algorithm depends upon the following...

1. Whether it is running on **Single** processor machine or **Multi** processor machine.
2. Whether it is a **32 bit** machine or **64 bit** machine.
3. **Read** and **Write** speed of the machine.
4. The amount of time required by an algorithm to perform **Arithmetic** operations, **logical** operations, **return** value and **assignment** operations etc.,
5. **Input** data

If any program requires a fixed amount of time for all input values then its time complexity is said to be Constant Time Complexity.

If the amount of time required by an algorithm is increased with the increase of input value then that time complexity is said to be Linear or variable Time Complexity.

$$\mathbf{T(P) = C + T_P(I)}$$

Where T (*P*) is the time taken by a program *P*, it is the sum of its compile time *C* and its run (or execution) time, $T_P(I)$

Fixed time requirements are Compile time (*C*), it is independent of instance characteristics and Variable time requirements are Run (execution) time T_P.

Consider the following program 1.4

```
int sum(int A[], int n)
{
   int sum = 0, i;
   for(i = 0; i < n; i++)
      sum = sum + A[i];
   return sum;
}
```

Program 1.4: Summing a list of Numbers

int sum(int A[], int n)	**Cost**	**Repetition**	**Total**
{	**Time require for line (units)**	**No. of Times Executed**	**Total Time required in worst case**
int sum = 0, i;	1	1	1
for(i = 0; i < n; i++)	1+1+1	1+(n+1)=n	2n+2
sum = sum + A[i];	2	n	2n
return sum;	1	1	1
}			

Figure 1.3: Time needed for program 1.4

1.4 Asymptotic Notation

Whenever we want to perform analysis of an algorithm, we need to calculate the complexity of that algorithm. But when we calculate complexity of an algorithm it does not provide exact amount of resource required. So instead of taking exact amount of resource we represent that complexity in a general form (Notation) which produces the basic nature of that algorithm. We use that general form (Notation) for analysis process.

Asymptotic notation of an algorithm is a mathematical representation of its complexity

In asymptotic notation, when we want to represent the complexity of an algorithm, we use only the most significant terms in the complexity of that algorithm and ignore least significant terms in the complexity of that algorithm (Here complexity may be Space Complexity or Time Complexity).

For example, consider the following time complexities of two algorithms...

- **Algorihtm 1 : $5n^2 + 2n + 1$**
- **Algorihtm 2 : $10n^2 + 8n + 3$**

Generally, when we analyze an algorithm, we consider the time complexity for larger values of input data (i.e. '**n**' value). In above two time complexities, for larger value of '**n**' the term in algorithm 1 '**2n + 1**' has least significance than the term '**$5n^2$**', and the term in algorithm 2 '**8n + 3**' has least significance than the term '**$10n^2$**'.

Here for larger value of '**n**' the value of most significant terms (**$5n^2$** and **$10n^2$**) is very larger than the value of least significant terms (**2n + 1** and **8n + 3**). So for larger value of '**n**' we ignore the least significant terms to represent overall time required by an algorithm. In asymptotic notation, we use only the most significant terms to represent the time complexity of an algorithm.

Majorly, we use THREE types of Asymptotic Notations and those are as follows...

1. **Big - Oh (O)**
2. **Big - Omega (Ω)**
3. **Big - Theta (Θ)**

1.4.1 Big - Oh Notation (O)

Big - Oh notation is used to define the **upper bound** of an algorithm in terms of Time Complexity.

That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity.

Big - Oh Notation can be defined as follows...Consider function **f(n)** the time complexity of an algorithm and **g(n)** is the most significant term. If **f(n) <= C g(n)** for all **n >= n_0**, **C > 0** and **n_0 >= 1**. Then we can represent **f(n)** as **O(g(n))**.

f(n) = O(g(n))

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis

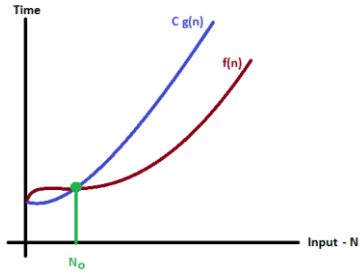

Figure 1.4

In above graph after a particular input value n_0, always C g(n) is greater than f(n) which indicates the algorithm's upper bound.

Example

Consider the following f(n) and g(n)...

f(n) = 3n + 2

g(n) = n

If we want to represent **f(n)** as **O(g(n))** then it must satisfy **f(n) <= C x g(n)** for all values of **C > 0** and **n_0>= 1**

f(n) <= C g(n)

⇒3n + 2 <= C n

Above condition is always TRUE for all values of **C = 4** and **n >= 2**.

By using Big - Oh notation we can represent the time complexity as follows...

3n + 2 = O(n)

1.4.2 Big - Omege Notation (Ω)

Big - Omega notation is used to define the **lower bound** of an algorithm in terms of Time Complexity.

That means Big - Omega notation always indicates the minimum time required by an algorithm for all input values. That means Big - Omega notation describes the best case of an algorithm time complexity.

Big - Omega Notation can be defined as follows...Consider function **f(n)** the time complexity of an algorithm and **g(n)** is the most significant term. If **f(n) >= C x g(n)** for all **n >= n_0**, **C > 0** and **n_0 >= 1**. Then we can represent **f(n)** as **Ω(g(n))**.

f(n) = Ω(g(n))

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis

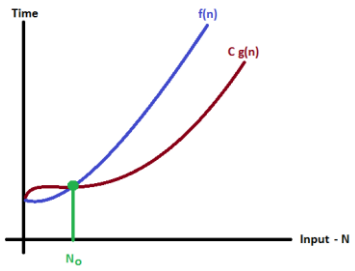

Figure 1.5

In above graph after a particular input value n_0, always C x g(n) is less than f(n) which indicates the algorithm's lower bound.

Example

Consider the following f(n) and g(n)...

f(n) = 3n + 2

g(n) = n

If we want to represent **f(n)** as **Ω(g(n))** then it must satisfy **f(n) >= C g(n)** for all values of **C > 0** and **n_0>= 1**

f(n) >= C g(n)

⇒3n + 2 <= C n

Above condition is always TRUE for all values of **C = 1** and **n >= 1**.

By using Big - Omega notation we can represent the time complexity as follows...

3n + 2 = Ω(n)

1.4.3 Big - Theta Notation (Θ)

Big - Theta notation is used to define the **average bound** of an algorithm in terms of Time Complexity.

That means Big - Theta notation always indicates the average time required by an algorithm for all input values. That means Big - Theta notation describes the average case of an algorithm time complexity.

Big - Theta Notation can be defined as follows...Consider function **f(n)** the time complexity of an algorithm and **g(n)** is the most significant term. If **C_1 g(n) <= f(n) >= C_2 g(n)** for all **n >= n_0**, **C_1, C_2 > 0** and **n_0 >= 1**. Then we can represent **f(n)** as **Θ(g(n))**.

f(n) = Θ(g(n))

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis

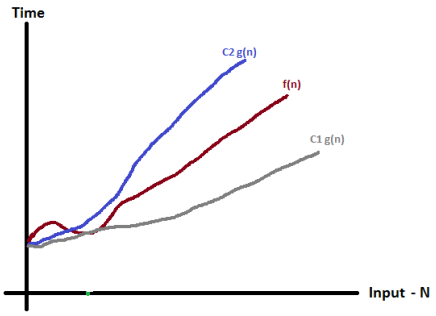

Figure 1.6

In above graph after a particular input value n_0, always **C_1 g(n)** is less than **f(n)** and **C_2 g(n)** is greater than **f(n)** which indicates the algorithm's average bound.

Example

Consider the following f(n) and g(n)...

f(n) = 3n + 2

g(n) = n

If we want to represent **f(n)** as **Θ(g(n))** then it must satisfy **C_1 g(n) <= f(n) >= C_2 g(n)** for all values of **C_1, C_2 > 0** and **n_0>= 1**
C_1 g(n) <= f(n) >= ⇒C_2 g(n)
C_1 n <= 3n + 2 >= C_2 n
Above condition is always TRUE for all values of **C_1 = 1, C_2 = 4** and **n >= 1**.
By using Big - Theta notation we can represent the time compexity as follows...
3n + 2 = Θ(n)

1.5 Programming Style

A programming style is a set of guidelines used to format programming instructions. It is useful to follow a style as it makes it easier for programmers to understand the code, maintain it, and assists in reducing the likelihood of introducing errors. Guidelines can be developed from coding conventions used in an organisation with variations of style occurring for different programming languages.

1. Clarity and simplicity of Expression: The programs should be designed in such a manner so that the objectives of the program are clear.
2. Naming: In a program, you are required to name the module, processes, and variable, and so on. Care should be taken that the naming style should not be cryptic and non-representative.
For Example: a = 3.14 * r * r
area of circle = 3.14 * radius * radius;
3. Control Constructs: It is desirable that as much as a possible single entry and single exit constructs used.
4. Information hiding: The information secure in the data structures should be hidden from the rest of the system where possible. Information hiding can decrease the coupling between modules and make the system more maintainable.
5. Nesting: Deep nesting of loops and conditions greatly harm the static and dynamic behavior of a program. It also becomes difficult to understand the program logic, so it is desirable to avoid deep nesting.
6. User-defined types: Make heavy use of user-defined data types like enum, class, structure, and union. These data types make your program code easy to write and easy to understand.
7. Module size: The module size should be uniform. The size of the module should not be too big or too small. If the module size is too large, it is not generally functionally cohesive. If the module size is too small, it leads to unnecessary overheads.
8. Module Interface: A module with a complex interface should be carefully examined.
9. Side-effects: When a module is invoked, it sometimes has a side effect of modifying the program state. Such side-effect should be avoided where as possible.

1.6 Refinement of Coding

Refinement of coding is the idea that software is developed by moving through the levels of abstraction, beginning at higher levels and, incrementally refining the software through each level of abstraction, providing more detail at each increment. At higher levels, the software is merely its design models; at lower levels there will be some code; at the lowest level the software has been completely developed

- Stepwise refinement of Coding: design a problem solution by

1. stating the solution at a high level
2. refining steps of the solution into simpler steps
3. repeating step 2, until steps are simple enough to execute

- Decompose based on function of each step
- Makes heavy use of pseudocode

Example:

- Problem: Print the count and average of a sequence of positive integers
- Initial Solution:
  1. Initialize data
  2. Get data
  3. Calculate results
  4. Output results
- Refinement:
  1. Initialize data
  2. get data
     - 2.1 loop
     - 2.2 get integer x
     - 2.3 exit when x < 1
     - 2.4 process x
  3. Calculate results
     - 3.1 avg = sum / count
  4. Output results
- Further refinement (of step 2.4):
  - 2.4.1 increase count by 1
  - 2.4.2 increase sum by x
- Process of refinement is complete when all steps can be easily programmed

1.7 Testing

Testing is the process of evaluating a system or its component(s) with the intent to find whether it satisfies the specified requirements or not.Testing is executing a system in order to identify any gaps, errors, or missing requirements in contrary to the actual requirements.

The process of software testing aims not only at finding faults in the existing software but also at finding measures to improve the software in terms of efficiency, accuracy and usability. It mainly aims at measuring specification, functionality and performance of a software program or application.

Software testing can be divided into two steps:

1. **Verification:** it refers to the set of tasks that ensure that software correctly implements a specific function.
2. **Validation:** it refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.

Verification: "Are we building the product right?"
Validation: "Are we building the right product?"

Types of software testing

Software Testing can be broadly classified into two types:

1. **Manual Testing:** Manual testing includes testing a software manually, i.e., without using any automated tool or any script. In this type, the tester takes over the role of an end-user and tests the software to identify any unexpected behaviour or bug. Testers use test plans, test cases, or test scenarios to test a software to ensure the completeness of testing.
2. **Automation Testing:** Automation testing, which is also known as Test Automation, is when the tester writes scripts and uses another software to test the product. This process involves automation of a manual process. Automation Testing is used to re-run the test scenarios that were performed manually, quickly, and repeatedly.

1.8 Introduction to Data Structure

- Data structure is an arrangement of data in computer's memory. It makes the data quickly available to the processor for required operations.
- It is a software artifact which allows data to be stored, organized and accessed.
- It is a structure program used to store ordered data, so that various operations can be performed on it easily.
- **For example,** if we have an employee's data like name 'ABC' and salary 10000. Here, 'ABC' is of String data type and 10000 is of Float data type.
- We can organize this data as a record like Employee record and collect & store employee's records in a file or database as a data structure like 'ABC' 10000, 'PQR' 15000, 'STU' 5000.
- Data structure is about providing data elements in terms of some relationship for better organization and storage.
- It is a specialized format for organizing and storing data that can be accessed within appropriate ways.

Types of Data Structure

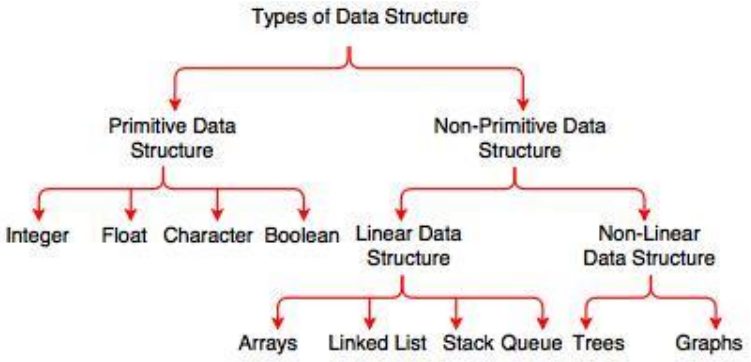

Fig. Types of Data Structure

A. Primitive Data Type

- Primitive data types are the data types available in most of the programming languages.
- These data types are used to represent single value.
- It is a basic data type available in most of the programming language.

Data type	Description
Integer	Used to represent a number without decimal point.

Float	Used to represent a number with decimal point.
Character	Used to represent single character.
Boolean	Used to represent logical values either true or false.

B. Non-Primitive Data Type

- Data type derived from primary data types are known as Non-Primitive data types.
- Non-Primitive data types are used to store group of values.

It can be divided into two types:

1. Linear Data Structure
2. Non-Linear Data Structure

1. Linear Data Structure

- Linear data structure traverses the data elements sequentially.
- In linear data structure, only one data element can directly be reached.
- It includes array, linked list, stack and queues.

Types	Description
Arrays	Array is a collection of elements. It is used in mathematical problems like matrix, algebra etc. each element of an array is referenced by a subscripted variable or value, called subscript or index enclosed in parenthesis.
Linked list	Linked list is a collection of data elements. It consists of two parts: Info and Link. Info gives information and Link is an address of next node. Linked list can be implemented by using pointers.
Stack	Stack is a list of elements. In stack, an element may be inserted or deleted at one end which is known as Top of the stack. It performs two operations: Push and Pop. Push means adding an element in stack and Pop means removing an element in stack. It is also called Last-in-First-out (LIFO).
Queue	Queue is a linear list of element. In queue, elements are added at one end called rear and the existing elements are deleted from other end called front. It is also called as First-in-First-out (FIFO).

2. Non-Linear Data Structure

- Non-Linear data structure is opposite to linear data structure.
- In non-linear data structure, the data values are not arranged in order and a data item is connected to several other data items.
- It uses memory efficiently. Free contiguous memory is not required for allocating data items.
- It includes trees and graphs.

Type	Description
Tree	Tree is a flexible, versatile and powerful non-linear data structure. It is used to represent data items processing hierarchical relationship between the grandfather

	and his children & grandchildren. It is an ideal data structure for representing hierarchical data.
Graph	Graph is a non-linear data structure which consists of a finite set of ordered pairs called edges. Graph is a set of elements connected by edges. Each elements are called a vertex and node.

1.9 Data Abstraction

The concept of representing important details and hiding away the implementation details is called data abstraction. This programming technique separates the interface and implementation.

The Data Type is basically a type of data that can be used in different computer program. It signifies the type like integer, float etc, the space like integer will take 4-bytes, character will take 1-byte of space etc.

An **abstract data type or ADT is a collection of data and a set of operations on the data** that **does not specify how the data is stored** or how the operations accomplish their functions.The abstract datatype is special kind of datatype, whose behavior is defined by a set of values and set of operations. The keyword "Abstract" is used as we can use these datatypes, we can perform different operations. But how those operations are working that is totally hidden from the user. The ADT is made of with primitive datatypes, but operation logics are hidden.

Some examples of ADT are Stack, Queue, List etc.

Let us see some operations of Stack ADT −

- isFull(), This is used to check whether stack is full or not
- isEmpry(), This is used to check whether stack is empty or not
- push(x), This is used to push x into the stack
- pop(), This is used to delete one element from top of the stack
- peek(), This is used to get the top most element of the stack
- size(), this function is used to get number of elements present into the stack