Unit1

Algorithm

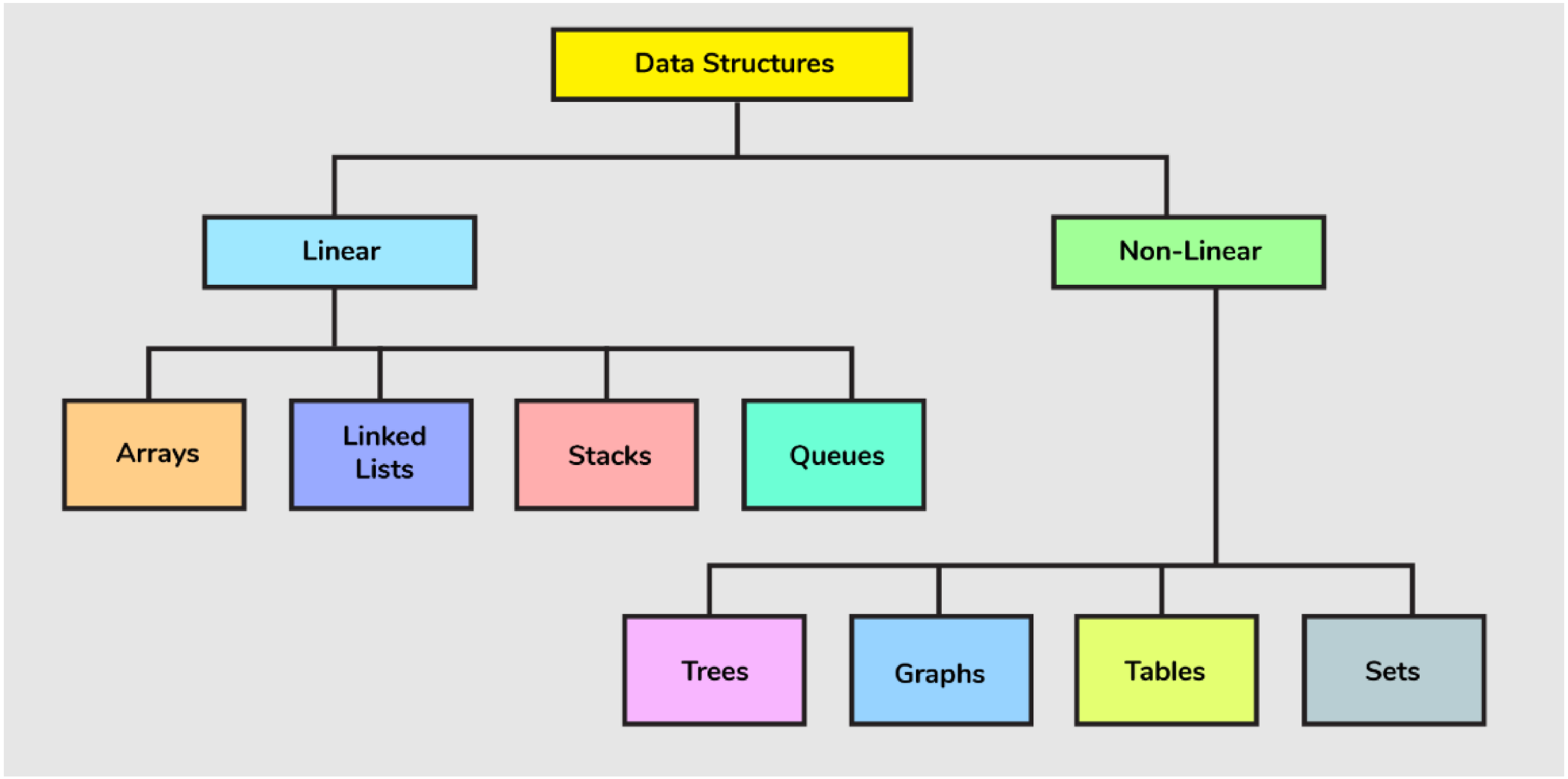
7. COURSE CONTENTS

UNIT NO.	Topics/Sub-Topics
I	**Introduction to data structure** 1.1 Linear & Non linear 1.2 Algorithm Basic Concepts 1.3 Introduction to Time and Space complexity of algorithms, Big O Notation and theta notations 1.4 Definition, implementation and notation of Array- Numerical and character 1.5 Basic operation such as addition, deletion , String operations
	[illegible]

- What is Data Structure?
- A data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.

- A data structure is not only used for organizing the data. It is also used for processing, retrieving, and storing data. There are different basic and advanced types of data structures that are used in almost every program or software system that has been developed. So we must have good knowledge about data structures.

Classification of Data Structure



- Linear Data Structure: Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure.
- Example: Array, Stack, Queue, Linked List, etc.

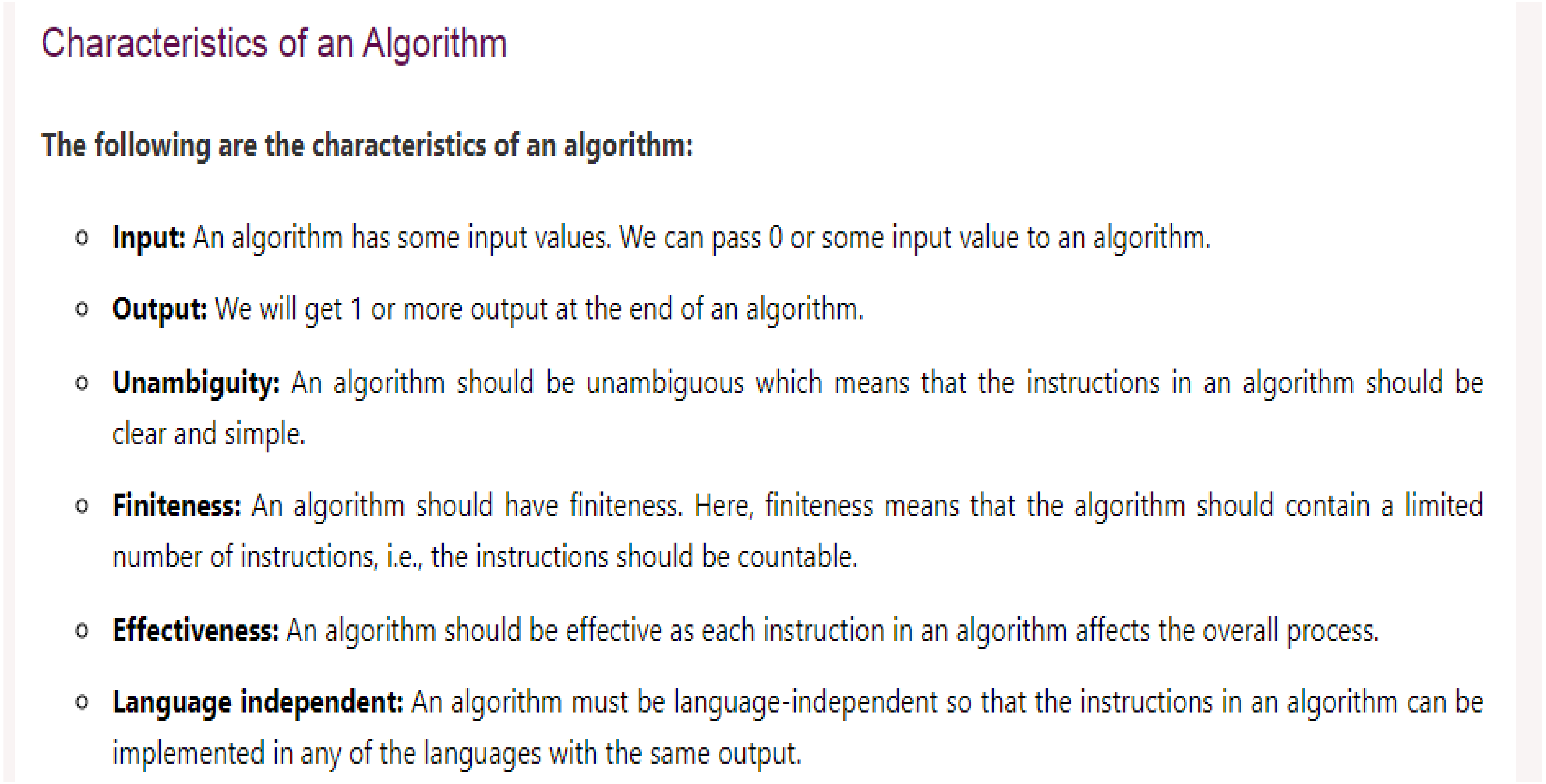
- Static Data Structure: Static data structure has a fixed memory size. It is easier to access the elements in a static data structure.
- Example: array.
- Dynamic Data Structure: In dynamic data structure, the size is not fixed. It can be randomly updated during the runtime which may be considered efficient concerning the memory (space) complexity of the code.
- Example: Queue, Stack, etc.

- Non-Linear Data Structure: Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. In a non-linear data structure, we can’t traverse all the elements in a single run only.
- Examples: Trees and Graphs.

Algorithm

- Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output.
- Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

- From the data structure point of view, following are some important categories of algorithms −
- Search − Algorithm to search an item in a data structure.
- Sort − Algorithm to sort items in a certain order.
- Insert − Algorithm to insert item in a data structure.
- Update − Algorithm to update an existing item in a data structure.
- Delete − Algorithm to delete an existing item from a data structure.



Characteristics of an Algorithm

The following are the characteristics of an algorithm:

- **Input:** An algorithm has some input values. We can pass 0 or some input value to an algorithm.
- **Output:** We will get 1 or more output at the end of an algorithm.
- **Unambiguity:** An algorithm should be unambiguous which means that the instructions in an algorithm should be clear and simple.
- **Finiteness:** An algorithm should have finiteness. Here, finiteness means that the algorithm should contain a limited number of instructions, i.e., the instructions should be countable.
- **Effectiveness:** An algorithm should be effective as each instruction in an algorithm affects the overall process.
- **Language independent:** An algorithm must be language-independent so that the instructions in an algorithm can be implemented in any of the languages with the same output.

Dataflow of an Algorithm

- **Problem:** A problem can be a real-world problem or any instance from the real-world problem for which we need to create a program or the set of instructions. The set of instructions is known as an algorithm.
- **Algorithm:** An algorithm will be designed for a problem which is a step by step procedure.
- **Input:** After designing an algorithm, the required and the desired inputs are provided to the algorithm.
- **Processing unit:** The input will be given to the processing unit, and the processing unit will produce the desired output.
- **Output:** The output is the outcome or the result of the program.

Why do we need Algorithms?

We need algorithms because of the following reasons:

- **Scalability:** It helps us to understand the scalability. When we have a big real-world problem, we need to scale it down into small-small steps to easily analyze the problem.
- **Performance:** The real-world is not easily broken down into smaller steps. If the problem can be easily broken into smaller steps means that the problem is feasible.

Algorithm example

Example

Let's try to learn algorithm-writing by using an example.

Problem – Design an algorithm to add two numbers and display the result.

```
Step 1 − START
Step 2 − declare three integers a, b & c
Step 3 − define values of a & b
Step 4 − add values of a & b
Step 5 − store output of step 4 to c
Step 6 − print c
Step 7 − STOP
```

Algorithms tell the programmers how to code the program. Alternatively, the algorithm can be written as –

```
Step 1 − START ADD
Step 2 − get values of a & b
Step 3 − c ← a + b
Step 4 − display c
Step 5 − STOP
```

- Two common approaches used in designing algorithms are the bottom-up and top-down methods.
- Top-Down Approach
- The top-down approach, also known as “divide and conquer,” begins by viewing the problem in its entirety.
- It starts with the main problem and breaks it down into smaller, more manageable sub problems.
- This approach is generally associated with the use of recursion and is prevalent in design techniques like dynamic programming and divide-and-conquer algorithms.

- For example, in the case of a sorting problem like Merge Sort, a top-down approach would start by splitting the entire array into smaller pieces, sorting these smaller pieces, and then merging them back together to create a sorted array.

- The bottom-up approach, on the other hand, starts with the simplest and most basic subproblems and gradually builds up solutions to larger subproblems.
- This technique is widely used in dynamic programming, where solutions to subproblems are typically stored in a table for easy access and to avoid recomputation.

- Take, for instance, the Fibonacci sequence, where each number is the sum of the two preceding ones. A bottom-up algorithm would start by computing the smallest values (base cases) of the sequence

Example 1: Write an algorithm to find the maximum of all the elements present in the array.

Follow the algorithm approach as below:

Step 1: Start

Step 2: Declare a variable max with the value of the first element of the array.

Step 3: Compare max with other elements using loop.

Step 4: If max < array element value, change max to new max.

Step 5: If no element is left, return or print max otherwise goto step 3.

Step 6: End

Example 2: Write an algorithm to find the average of 3 subjects.

- Step 1: Start
- Step 2: Declare and Read 3 Subject, let’s say S1, S2, S3
- Step 3: Calculate the sum of all the 3 Subject values and store result in Sum variable (Sum = S1+S2+S3)
- Step 4: Divide Sum by 3 and assign it to Average variable.
- (Average = Sum/3)
- Step 5: Print the value of Average of 3 Subjects
- Step 6: End