Design & Analysis of Algorithm

Course Outline

UNIT I: Introduction

- Algorithm, Psuedo code, Performance Analysis- Space complexity, Time complexity, Asymptotic Notation- Big oh notation, Omega notation, Theta notation with numerical, different algorithm design techniques, recurrence relation, solving methods: substitution, recursion tree, master theorem with numerical.

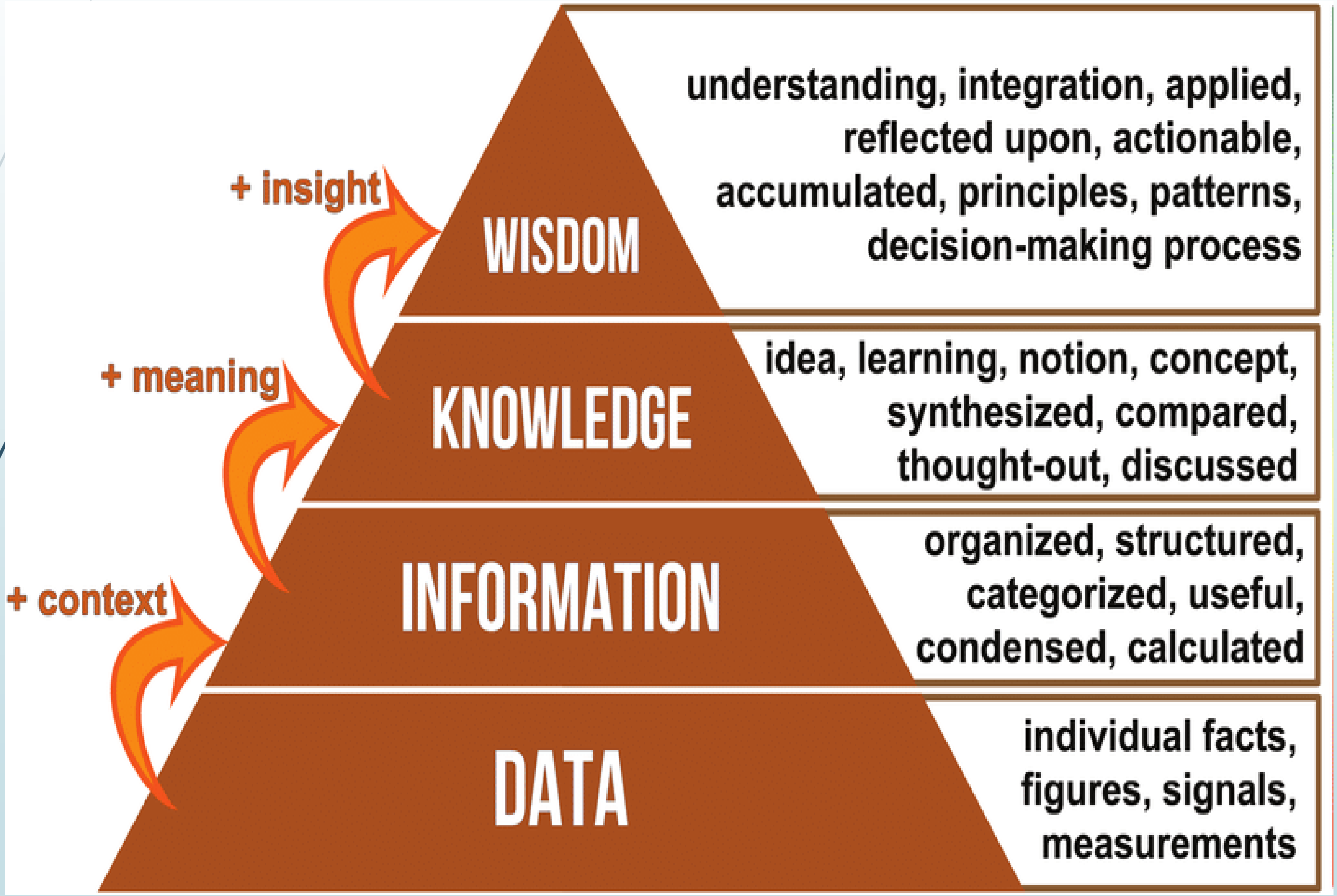
Introduction

Program

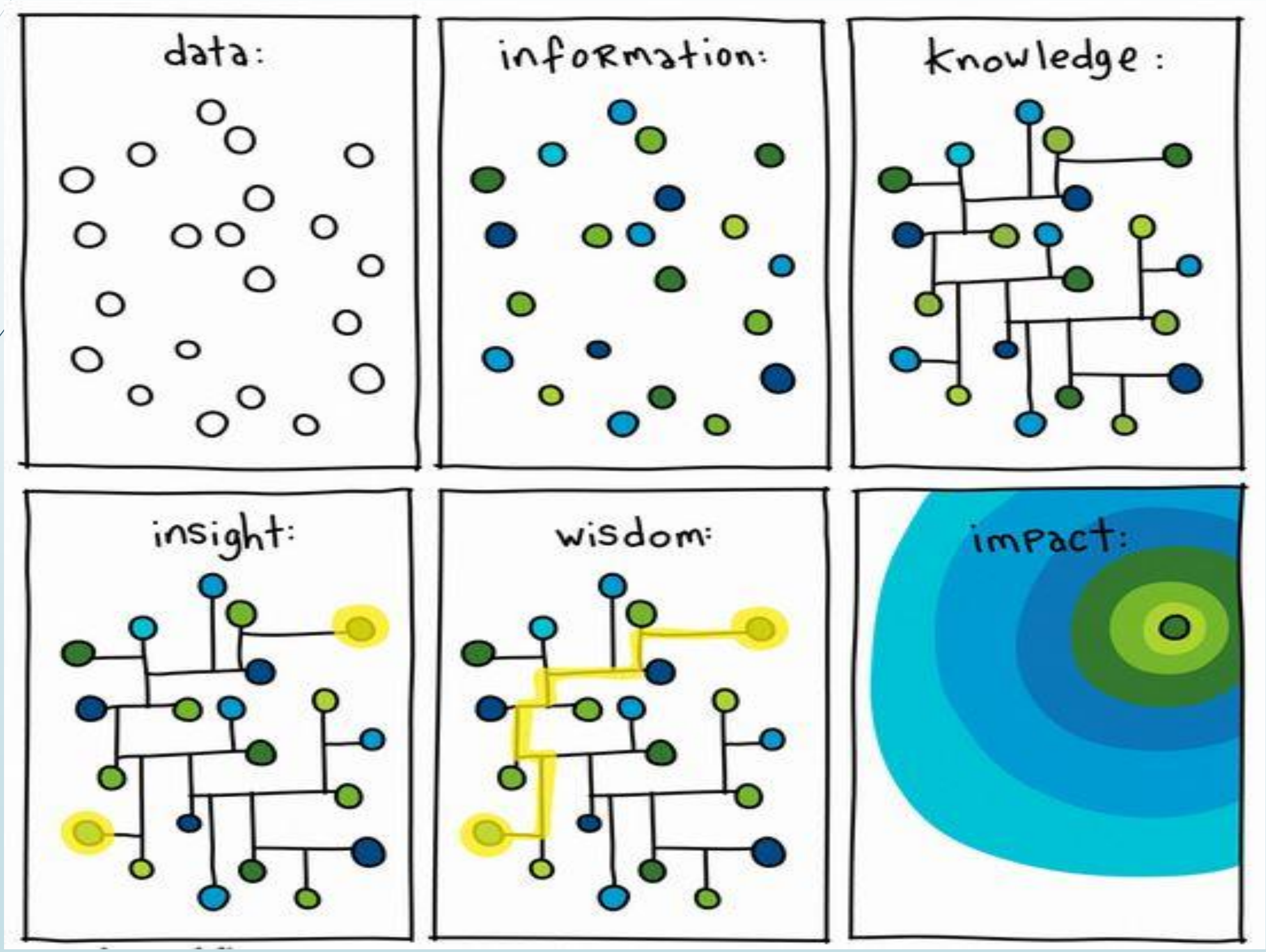
Algorithm

Data Structure

Data at Glance



Data at Glance



Data Structure

- **Data structure** is representation of the **logical relationship** existing between individual elements of data.

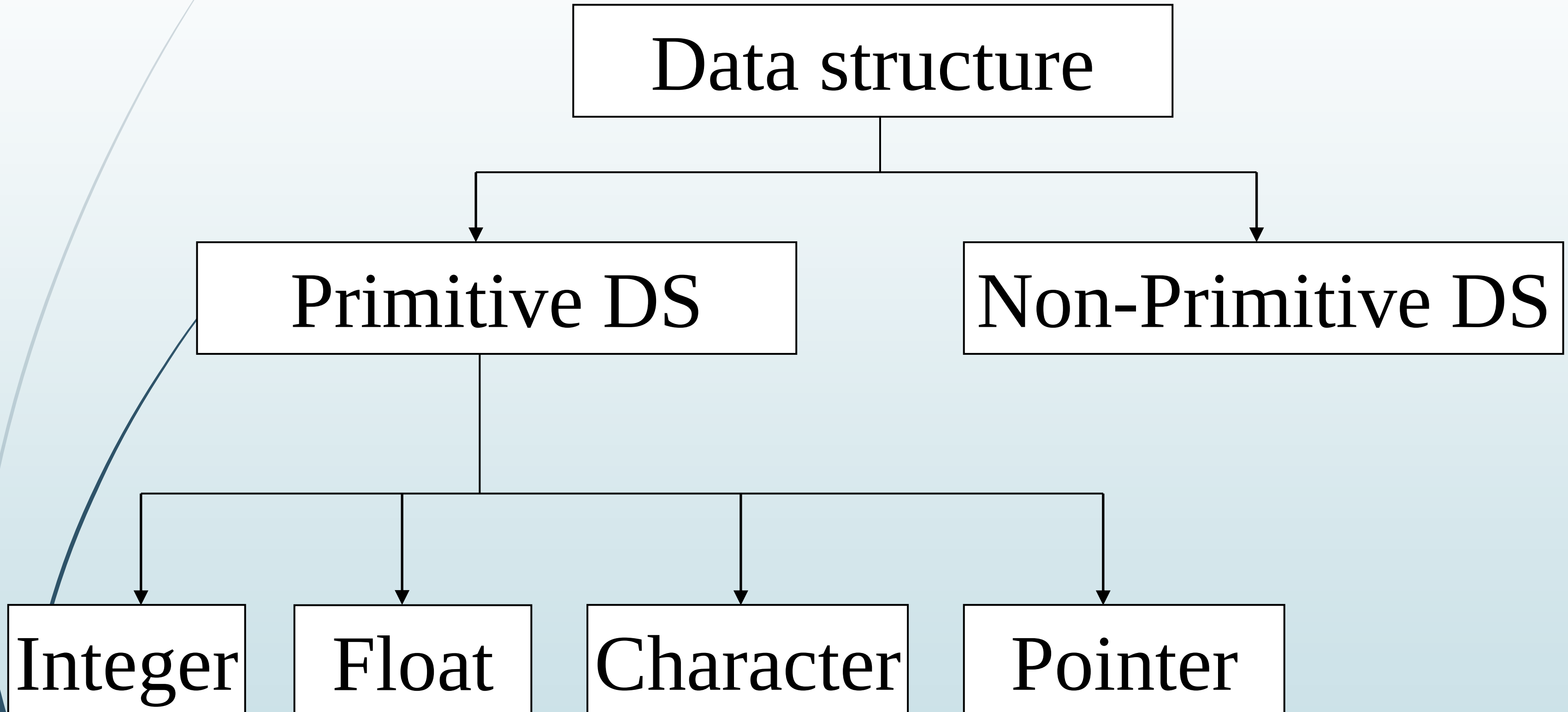
- In other words, a **data structure** is a way of organizing all data items that considers not only the elements stored but also their relationship to each other.

- Data structure affects the design of both **structural & functional aspects** of a program.

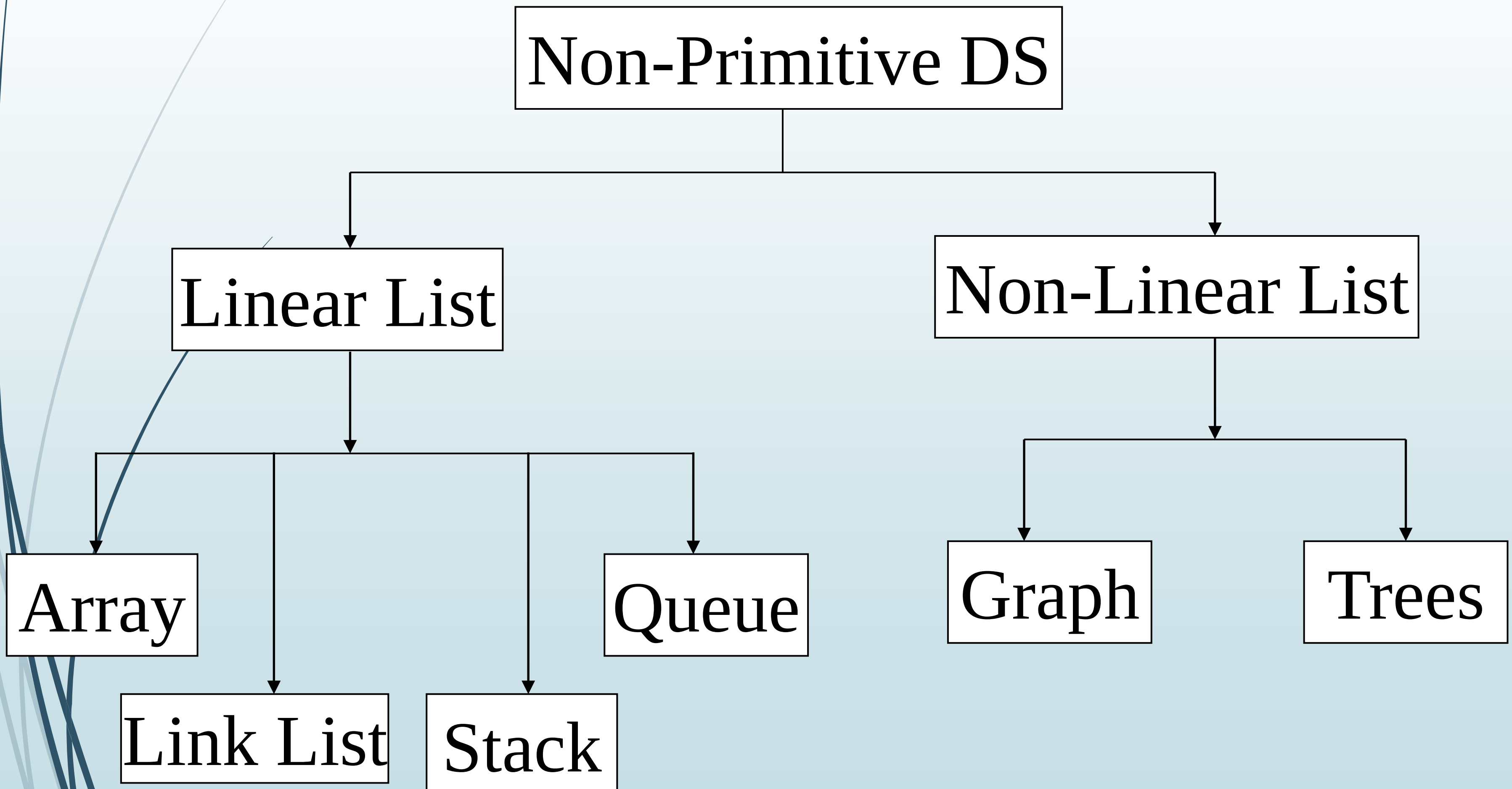
Classification of Data Structure

- Data structure are normally divided into two broad categories:
  - **Primitive Data Structure**
  - **Non-Primitive Data Structure**

Classification of Data Structure



Classification of Data Structure



Primitive Data Structure

- There are basic structures and directly operated upon by the machine instructions.
- In general, there are different representation on different computers.
- Integer, Floating-point number, Character constants, string constants, pointers etc., fall in this category.

Non-Primitive Data Structure

- There are more sophisticated data structures.
- These are derived from the primitive data structures.
- The non-primitive data structures emphasize on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items.
- Lists, Stack, Queue, Tree, Graph are example of non-primitive data structures.
- The design of an efficient data structure must take operations to be performed on the data structure.

Non-Primitive Data Structure

- The most commonly used operation on data structure are broadly categorized into following types:
  - Create
  - Selection
  - Updating
  - Searching
  - Sorting
  - Merging
  - Destroy or Delete

Different between them

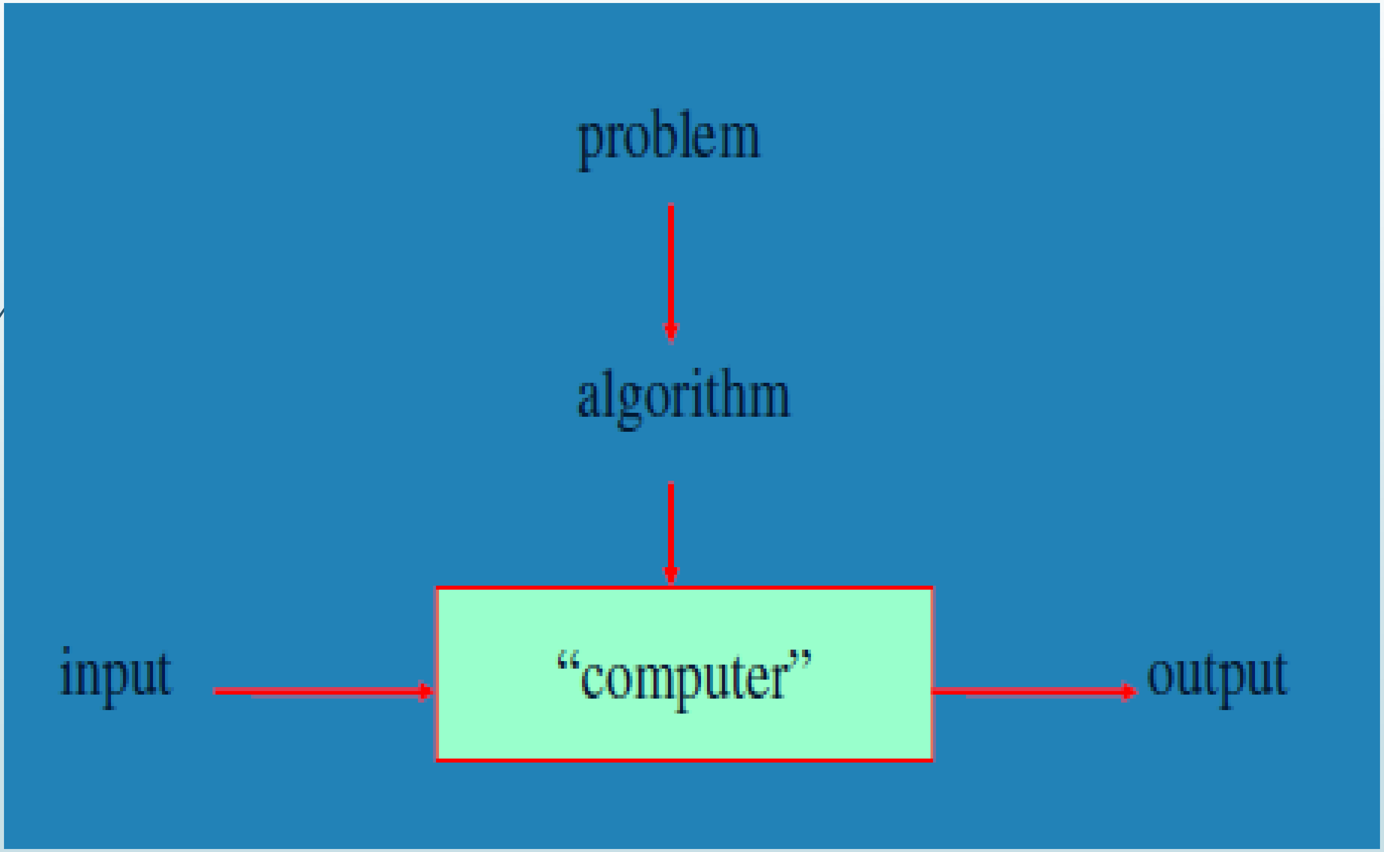
- A primitive data structure is generally a basic structure that is usually built into the language, such as an integer, a float.

- A non-primitive data structure is built out of primitive data structures linked together in meaningful ways, such as a or a linked-list, binary search tree, AVL Tree, graph etc.

Algorithm

- A algorithm is nothing but a step by step procedure to solve a particular problem.
- Algorithm is a set of instruction written to carry out certain tasks & the data structure is the way of organizing the data with their logical relationship retained.
- To develop a program of an algorithm, we should select an appropriate data structure for that algorithm.
- Therefore algorithm and its associated data structures from a program.

Algorithm



Properties of Algorithm

- **Input:** zero or more quantities externally supplied
- **Output:** at least one quantity is produced
- **Definiteness:** Each instruction must be clear and unambiguous.
- **Finiteness:** In all cases algorithm must terminate after finite number of steps.
- **Effectiveness:** each instruction must be sufficiently basic.

Pseudo Code

- **Pseudo Code** is an artificial and informal language that helps programmers develop algorithms. Pseudocode is a **"text-based"** detail (algorithmic) design tool.

- The rules of **Pseudo Code** are reasonably straightforward. All statements showing **"dependency"** are to be indented. These include while, do, for, if, switch. Examples below will illustrate this notion.

Pseudo Code for Expressing Algorithm

- **Algorithm heading-** It consists of name of algorithm, problem description, input, output

- **Algorithm body-** It consists of logical body of the alg. By using various programming constructs and assignment statements

Psuedo Code Outlook

- **INPUT** – indicates a user will be inputting something
- **OUTPUT** – indicates that an output will appear on the screen
- **WHILE** – a loop (iteration that has a condition at the beginning)
- **FOR** – a counting loop (iteration)
- **REPEAT – UNTIL** – a loop (iteration) that has a condition at the end
- **IF – THEN – ELSE** – a decision (selection) in which a choice is made
- Any instructions that occur inside a selection or iteration are usually indented

Example 1: Add Two Numbers

1. **BEGIN**
2. NUMBER s1, s2, sum
3. OUTPUT("Input number1:")
4. INPUT s1
5. OUTPUT("Input number2:")
6. INPUT s2
7. sum=s1+s2
8. OUTPUT sum
9. **END**

Example 2: Calculate Area and Perimeter of Rectangle

1. **BEGIN**
2. NUMBER b1,b2,area,perimeter
3. INPUT b1
4. INPUT b2
5. area=b1*b2
6. perimeter=2*(b1+b2)
7. OUTPUT area
8. OUTPUT perimeter
9. **END**

Example 3: Find Area Of Circle using Radius

1. **BEGIN**
2. NUMBER r, area
3. INPUT r
4. area=3.14*r*r
5. OUTPUT area
6. **END**

Example 4: Find Perimeter Of Circle using Radius

1. **BEGIN**
2. NUMBER r, perimeter
3. INPUT r
4. perimeter=2*3.14*r
5. OUTPUT perimeter
6. **END**

To Do List

1. Find the biggest of three (3)

2. Find Sum of Natural Numbers (1 to 100).

3. Read 10 numbers and find sum of even numbers.

Find the biggest of three (3)

1. **BEGIN**
2. NUMBER num1,num2,num3
3. INPUT num1
4. INPUT num2
5. INPUT num3
6. IF num1>num2 AND num1>num3 THEN
7.   OUTPUT num1+ "is higher"
8. ELSE IF num2 > num3 THEN
9.   OUTPUT num2 + "is higher"
10. ELSE
11.   OUTPUT num3+ "is higher"
12. ENDIF
13. **END**

Find Sum of Natural Numbers (1 to 100)

1. **BEGIN**
2. NUMBER counter, sum=0
3. FOR counter=1 TO 100 STEP 1 DO
4. sum=sum+counter
5. ENDFOR
6. OUTPUT sum
7. **END**

Read 10 numbers and find sum of even numbers

```
BEGIN
NUMBER counter, sum=0, num
FOR counter=1 TO 10 STEP 1 DO
    OUTPUT "Enter a Number"
    INPUT num
    IF num % 2 == 0 THEN
        sum=sum+num
    ENDIF
ENDFOR
OUTPUT sum
END
```

Complexity of algorithm

- Space Complexity
- Time Complexity

Space Complexity

- The space complexity of an algorithm is the amount of memory it needs to run to completion.
- The space includes instruction space, variable space, and space for constant.
- The space needed by each of these programs is seen to be the sum of the two components
  a) a fixed part that is independent of the characteristics (number/size) of the input and outputs.
  b) a variable part that consists of the space needed by component variable whose size is dependent on the particular problem instance being solved like the space needed by referenced variables.
- The space requirement S(P) of any algorithm P may therefore be written as S(P)=c+Sp (instance characteristics) ,where c is a constant.

Time Complexity

- The time complexity of an algorithm is the amount of computer (CPU) time it needs to run to completion.

- The time T(P) taken by a program P is the sum of the compile time and the run(execution) time.

Growth of Functions and Asymptotic Notation

- When we study algorithms, we are interested in characterizing them according to their efficiency.
- We are usually interesting in the order of growth of the running time of an algorithm, not in the exact running time. This is also referred to as the *asymptotic running time*.
- We need to develop a way to talk about rate of growth of functions so that we can compare algorithms.
- *Asymptotic notation* gives us a method for classifying functions according to their rate of growth.

Big-O Notation

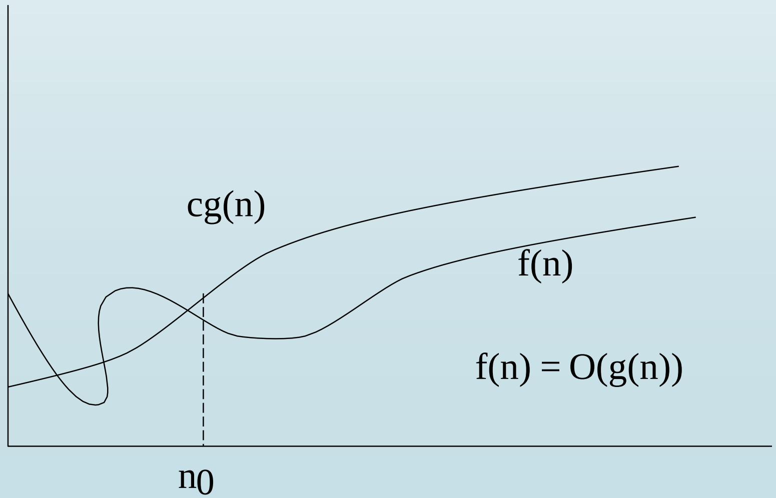
- **Definition:** $f(n) = O(g(n))$ if there are two positive constants c and n_0 such that

$|f(n)| \leq c\,|g(n)|$ for all $n \geq n_0$

- If $f(n)$ is nonnegative, we can simplify the last condition to

$0 \leq f(n) \leq c\, g(n)$ for all $n \geq n_0$

- We say that "$f(n)$ is big-O of $g(n)$."
- As n increases, $f(n)$ grows no faster than $g(n)$. In other words, $g(n)$ is an *asymptotic upper bound* on $f(n)$.



Big-Ω notation

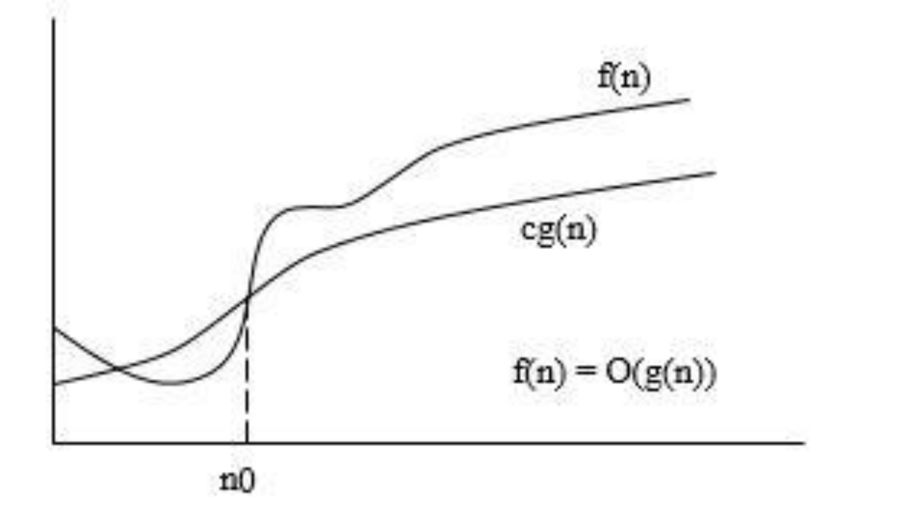
- **Definition:** $f(n) = \Omega(g(n))$ if there are two positive constants c and n_0 such that

$|f(n)| \geq c\ |g(n)|$ for all $n \geq n_0$

- If $f(n)$ is nonnegative, we can simplify the last condition to

$0 \leq c\ g(n) \leq f(n)$ for all $n \geq n_0$

- We say that "$f(n)$ is omega of $g(n)$."
- As n increases, $f(n)$ grows no slower than $g(n)$. In other words, $g(n)$ is an *asymptotic lower bound* on $f(n)$.



Big-Θ notation

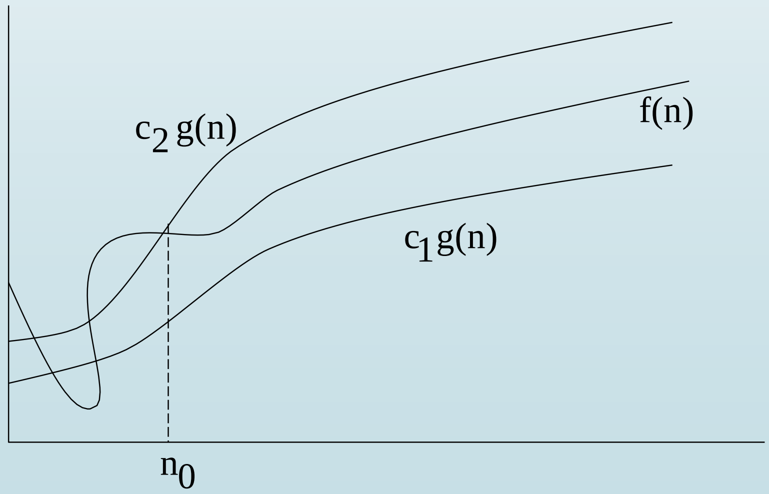
- **Definition:** $f(n) = \Theta(g(n))$ if there are three positive constants c_1, c_2 and n_0 such that

$c_1|g(n)| \le |f(n)| \le c_2|g(n)|$ for all $n \ge n_0$

- If $f(n)$ is nonnegative, we can simplify the last condition to

$0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$

- We say that "$f(n)$ is theta of $g(n)$."
- As n increases, $f(n)$ grows at the same rate as $g(n)$. In other words, $g(n)$ is an *asymptotically tight bound* on $f(n)$.



Asymptotic Bounds and Algorithms

- In general, it may be very difficult to determine the exact running time.
- Thus, we will try to determine a bounds without computing the exact running time.
  - **Example:** What is the complexity of the following algorithm?

    for (i = 0; i < n; i ++)

    for (j = 0; j < n; j ++)

    a[i][j] = b[i][j] * x;

    Answer: $O(n^2)$

Rates of Growth

TABLE 2.1 Values (some approximate) of several functions important for analysis of algorithms

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Rates of Growth

constant	$\theta(n^0) = \theta(1)$
logarithmic	$\theta(\lg n)$
linear	$\theta(n)$
<"en log en">	$\theta(n \lg n)$
quadratic	$\theta(n^2)$
cubic	$\theta(n^3)$
polynomial	$\theta(n^k)$, $k \geq 1$
exponential	$\theta(a^n)$, $a > 1$

Growth Rate Increasing