UNIVERSITY COLLEGE OF APPLIED SCIENCES

ALGORITHMS AND DATA STRUCTURES
ITDEP 2302

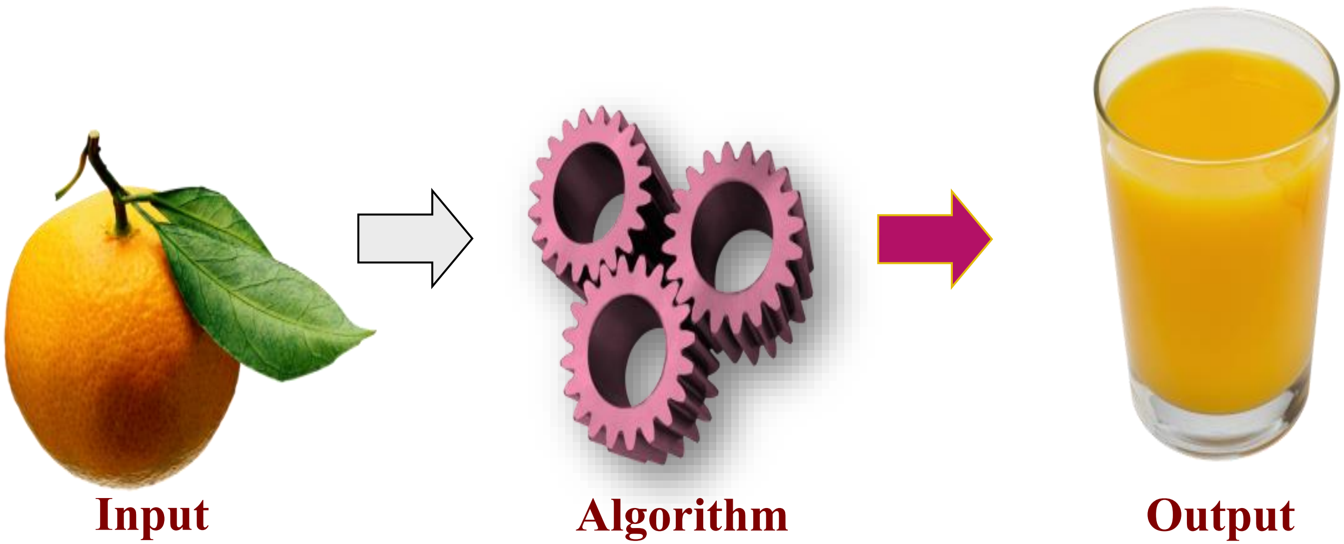
CHAPTER 1

ANALYSIS OF ALGORITHMS

Dr. Jawad Y. I. Alzamily

2022-2023

Analysis of Algorithms



What is an Algorithm?

- **What is an Algorithm?**

An algorithm is a set of well-defined instructions to solve a particular problem. It takes a set of input(s) and produces the desired output.

For example :

An algorithm to add two numbers:

1. Take two number inputs
2. Add numbers using the + operator
3. Display the result

Qualities of a Good Algorithm

Qualities of a Good Algorithm

- Input and output should be **defined precisely**.
- Each step in the algorithm should be clear and **unambiguous**.
- Algorithms should be most effective among many different ways to **solve a problem**.
- An algorithm shouldn't include computer code. Instead, the algorithm should be written in such a way that it can be used in different programming languages.

Algorithm Examples

Algorithm 1: Add two numbers entered by the user

- Step 1: Start
- Step 2: Declare variables num1, num2 and sum.
- Step 3: Read values num1 and num2.
- Step 4: Add num1 and num2 and assign the result to sum.
- sum←num1+num2
- Step 5: Display sum
- Step 6: Stop

Algorithm Examples

Algorithm 2: Find the largest number among three numbers

- Step 1: Start
- Step 2: Declare variables a,b and c.
- Step 3: Read variables a,b and c.
- Step 4: If a > b
-     If a > c
-       Display a is the largest number.
-      Else
-      Display c is the largest number.
-   Else
-   If b > c
-     Display b is the largest number.
-     Else
-     Display c is the greatest number.
- Step 5: Stop

Algorithm Examples

Algorithm 3: Find the factorial of a number

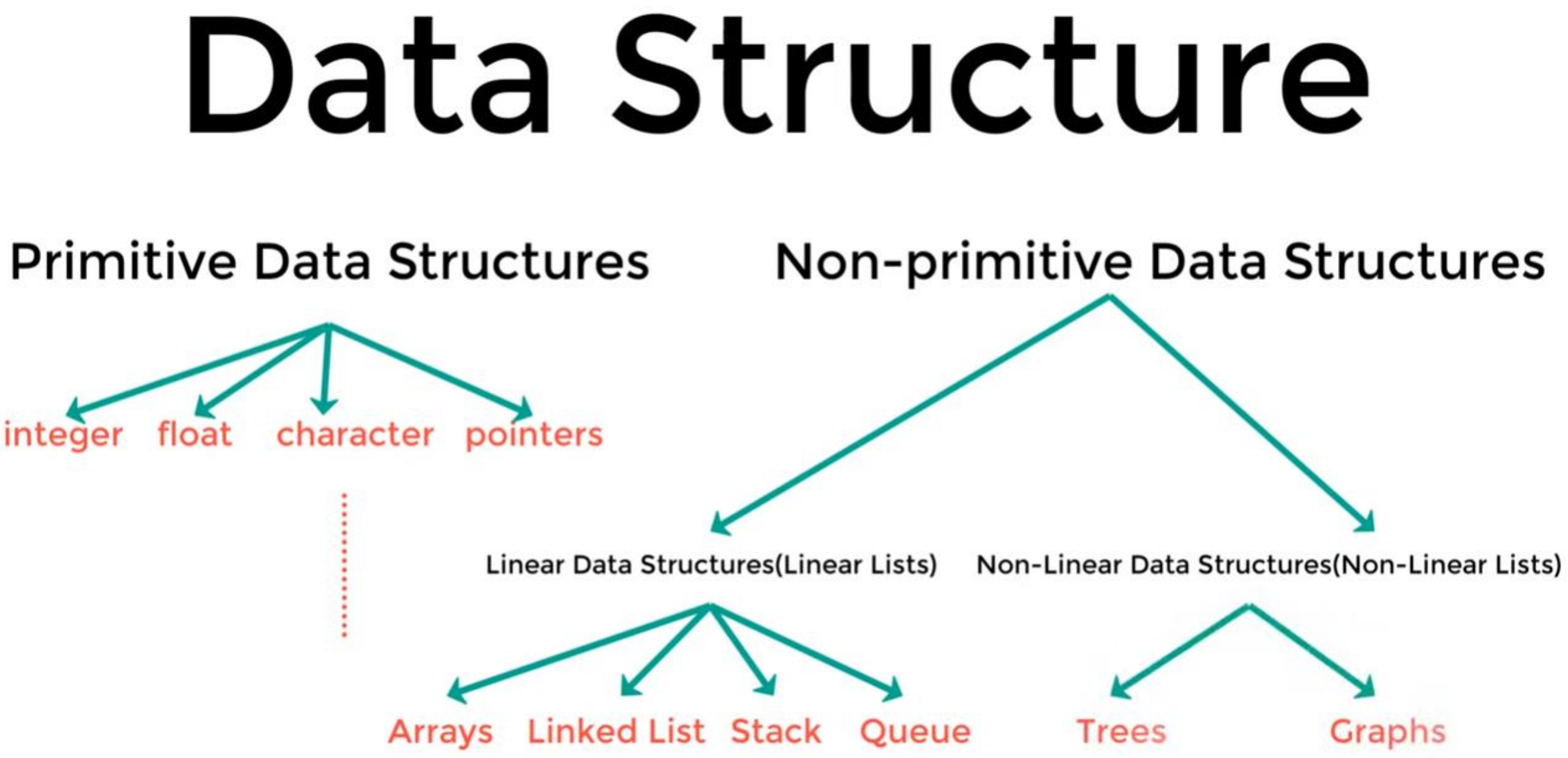
- Step 1: Start
- Step 2: Declare variables n, factorial and i.
- Step 3: Initialize variables
- factorial ← 1
- i ← 1
- Step 4: Read value of n
- Step 5: Repeat the steps until i = n
- 5.1: factorial ← factorial*i
- 5.2: i ← i+1
- Step 6: Display factorial
- Step 7: Stop

What are Data Structures

- **What are Data Structures?**

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.

Types of Data Structure



Linear Vs Non-linear Data Structures

Non Linear Data Structures	Linear Data Structures
The data items are arranged in non-sequential order (hierarchical manner).	The data items are arranged in sequential order, one after the other.
The data items are present at different layers.	All the items are present on the single layer.
It requires multiple runs. That is, if we start from the first element it might not be possible to traverse all the elements in a single pass.	It can be traversed on a single run. That is, if we start from the first element, we can traverse all the elements sequentially in a single pass.
Different structures utilize memory in different efficient ways depending on the need.	The memory utilization is not efficient.
Time complexity remains the same.	The time complexity increase with the data size.
Example: Tree, Graph, Map	Example: Arrays, Stack, Queue

Why Data Structure?

Why Data Structure?

- Knowledge about data structures help you understand the working of each data structure. And, based on that you can select the right data structures for your project.
- This helps you write memory and time efficient code.

Complexity

- **Analysis of algorithms:** an investigation of an algorithm's efficiency with respect to two resources:

  1. **running time** (Time efficiency or complexity)
  2. **memory space** (space efficiency or complexity)

  Efficiency can be studied in precise quantitative terms unlike **simplicity** ang **generality**

Running Time

- The running time of an algorithm typically grows with the input size.

Algorithms efficiency depends on form of input:

- **Best case:** minimum over inputs of size *n*
- **Worst case:** maximum over inputs of size *n*
- **Average case:** "average" over inputs of size *n*,

Best-case, Average-case, Worst-case

- **Find (3) is the Best Case**

3	6	8	4	12

- **Find (8) is the Average Case**

3	6	8	4	12

- **Find (12) is the Worst Case**

3	6	8	4	12

Best-case, Average-case, Worst-case

best case	Omega Notation, Ω
average case	theta notation, Θ
worst case	Big O notation, O

Best-case, Average-case, Worst-case

Why Worst case is the most important :

Best-case is not representative.

Average-case analysis:

is ideal, but difficult to perform, because it is hard to determine the relative probabilities and distributions of various input instances for many problems.

Worst-case is not representative

but worst-case analysis is very useful. You can show that the algorithm will never be slower than the worst-case.

Time Complexity

* , / , % , ^ , + , -

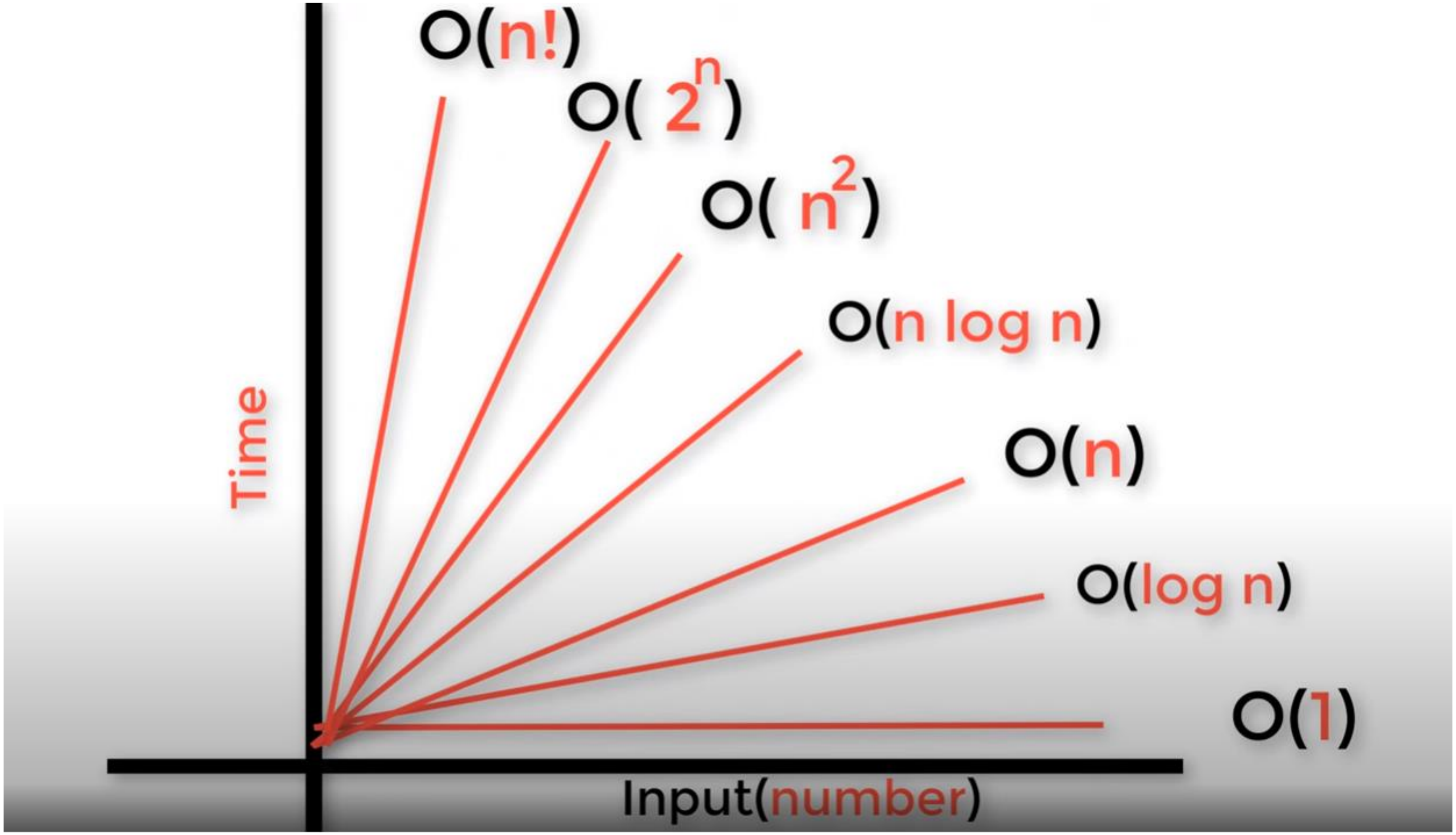
++ , --

+= , -= , /= , *=

if, else , ifelse

constant time = 1

Time Complexity



Time Complexity

Example 1:

```
int i;
i=0;          1
for (i=0; i<n; i++)        n
{  Console.Write( i );   }
```

1+n = O(n)

Time Complexity

Example 2:

```
int i,j;
i=0;          1
for (i=0; i<n; i++)        n
{  Console.Write( i );   }
for (j=0; j<n; j++)        n
for (int k=0; k<n; k++)    n
for (int m=1;m<n; m++)     n
```

1+n+(n*n*n) = 1+n+n^3 = O(n^3)

Time Complexity

Example 3:

```
int i;
i=1;                1
for (i=1; i<n; i=i*2)        log2 n
{  Console.Write( i );   }
```

$1 + \log_2 n = O(\log_2 n)$

Time Complexity

Example 4:

`i=1;  j=1;` 1

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

`for (j=1; j<n; j=j/3)` $\log_2 n$

`{  Console.Write( i +j );   }`

$1 + n * \log_2 n = O(n \log_2 n)$

Time Complexity

Example 5:

```
for (i=0; i<n; i++)        n
for (j=0; j<n; j++)        n
for (int k=1; k<n; k=k*2) log2 n
{  Console.Write( i +j +k);   }
```

$(n*n*\log_2 n) = n^2 * \log_2 n = O(n^2 \log_2 n)$

Time Complexity

Example 6:

```
for (i=1; i<n/2; i++)        n/2
for (j=1; j<n; j=j*2)      log2 n
for (int k=1; k<n; k=2*k) log2 n
{  Console.Write( i +j +k);   }
```

$(n/2 * \log_2 n \ * \log_2 n\ ) = n * \log_2 n^2 =$

$O(n \log_2 n^2)$