Excellence • Leadership • Service

CITY GOVERNEMENT OF SAN PABLO

DALUBHASAAN NG LUNSOD NG SAN PABLO

CHED Recognized Local College

TESDA Recognized Programs

ALCU Commission on Accreditation – Level 1 Reaccredited

Member, Association of Local Colleges and Universities

Member, Local Colleges and Universities Athletic Association, Inc.

MODULE

Data Structures and Algorithm

CC214

ACADEMIC YEAR 2024-2025

Prepared by:

BRYAN MAY Q. BOONGALING
Course Instructor

GUIDE ON HOW TO USE THE LEARNING MODULE

A. For Faculty

1. The Course Instructor must review and check the list of the enrolled learners in the course together with the contact information the learners have provided such as cell phone number and email address.

2. Before distributing the Learning Modules make sure it has been checked and reviewed by the Program Chair, Department Chair and the College Dean and with the recommending approval from the CLAMDev (Center for Learning and Assessment Materials Development).

3. Once it has been checked and approved, the Course Instructor must provide each learner a softcopy of the Learning Modules through a provided LMS (Learning Management System); any social media platforms will also be considered.

4. It is the responsibility of the Course Instructor to ensure that all of the learners have downloaded a softcopy of the Learning Modules.

5. Encourage the class that it is way better if they will print a hardcopy of the Learning Modules; but consider this as an option if they lack of resources.

6. Orient the class on how to properly use the Learning Modules and explain the content especially the lesson proper, the activities and exercises, the grading rubrics and criteria, and the submission format.

7. The Course Instructor must be open to any concerns and inquiries of each learner regarding the lessons and activities in the Learning Modules inclusively.

8. Always provide each learner a constructive feedback and a key to correction for every quizzes, activities and exercises they have done. Motivate them!

9. Give each learner enough time to accomplish and submit the requirements or any activities; especially not all learners have sufficient resources. Be considerate.

10. Keep Safe! Stay Healthy!
    Enjoy Teaching! Fist Bumps! XDD

GUIDE ON HOW TO USE THE LEARNING MODULE

B. For Learners

1. Each learners enrolled in the course will be provided by their Course Instructor a softcopy of the Learning Modules through an LMS (preferably Google Classroom) or any social media platforms available such as Facebook Messenger.

2. It is recommended for each learner to print a hardcopy of the Learning Modules for ease and clear understanding of the lessons; but again consider it as an option.

3. Different set of Learning Modules will be sent to each learner: weekly or monthly; depending on their Course Instructors.

4. Each Learning Modules contains different topics and lessons; and at the end of each lesson the learner must accomplished different activities and exercises.

5. The activities varies to every topics covered on the Learning Modules, this activities consists of Academic and Life Activity. Academic Activities contains Quizzes and Individual or Group Activities.

6. For every group works and activities provided in the Learning Modules, make sure to work well with your teammates and collaborate with them to accomplish the work assigned smoothly.

7. When submitting an activity or exercises online, learners must follow the mode and format of submissions included in the Learning Modules.

8. Learners must ensure a good time management and prioritization regarding the tasks and activities given to them. This is a very helpful reminder since classes are all done online.

9. If learners have any questions and queries regarding the course, lessons or any activities, do not hesitate to contact and ask your Course Instructor. Always remember that we are here not only to teach you but also to help you.

10. Keep Safe! Stay Healthy!
    Enjoy Learning! Fist Bumps! XDD

FOREWORD

Data Structures and Algorithms is a three (3) unit course prefer for the second year students of the Bachelor of Science in Information Technology (BSIT) and Bachelor of Science in Information Systems (BSIS) from the College of Computer Studies and Technology in the Pamantasan ng Lungsod ng San Pablo.

This course specifically focuses on the basic and essential topics in Data Structures and Algorithm including: Array, Linked Lists, Stack, Queue, Graph and Tree; Sorting, Searching, and Hashing Algorithms is also covered. This will engage learners the ability to analyze, design, apply, and use Data Structures and Algorithms to solve IT problems and define appropriate solutions in IT such as designing efficient computer programs that will cope with the complexity of actual applications.

Keep Safe . . God Bless . .

MODULE FOR DATA STRUCTURES AND ALGORITHM

Credits	: **3 Units** (3 Hours Laboratory, 2 Hours Lecture)
Pre-Requisite	: **CC123 – Computer Programming 2**

Lesson Title:

Lesson 1 – Preliminaries of Algorithm

A. Categories of Algorithms
B. Characteristics of Algorithms
C. Algorithm Analysis and Complexities
D. Asymptotic Analysis and Notations
E. Recursion
F. Pseudocode and Flowchart

Lesson Objective:

At the end of the module, the learners will be able to:

1. Express and discuss pre – existing knowledge of Algorithms.
2. Provide real – world examples of Algorithms from daily life activities.
3. Work in groups to create and implement Algorithms.
4. Create an Algorithm to help solve a real – world problem.
5. Produce Algorithms in Pseudocode or Flowcharts to solve computer – based problems.

Course Text Book:

Puntambekar, A. A. (2019). Data Structures. Chapter 1 – Recursion and Linear Search.

Chaudhuri, A. B. (2020). Flowchart and Algorithm Basics. Chapter 1 – Introduction to Programming (Flowcharting and Algorithms).

References:

Puntambekar, A. A. (2019). Data Structures Using C: A Conceptual Approach (1st Edition). Pune, India: Technical Publications. (Original work published 2014).

Delfinado, C. J. (2016). Data Structures and Algorithms. Quezon City, Philippines: C & E Publishing, Inc.

Chaudhuri, A. B. (2020). Flowchart and Algorithm Basics: The Art of Programming. Mercury Learning and Information. Retrived from https://bookrar.org/flowchart-and-algorithm-basics-the-art-of-programming/

Karumanchi, N. (2017). Data Structures and Algorithms Made Easy. 5th Edition. CareerMonk Publications. Retrived from https://www.pdfdrive.com/data-structures-and-algorithms-made-easy-data-structures-and-algorithmic-puzzles-e196527822.html

Tutorials Point (I) Pvt. Ltd. (2016). Data Structures and Algorithms. Tutorials Point: Simply Easy Learning. Retrieved from http://index-of.es/Varios-2/Data%20Structures%20and%20Algorithms%20Tutorial.pdf

Jadeja, P. (n.d.). Introduction to Data Structure. Darshan, Institute of Engineering and Technology. Retrieved from http://iykelnhub.org/media/document/data_structure _1.pdf

CS Dojo – YouTube Channel (2018). Data Structures and Algorithms (Videos 1 - 13). Retrieved from https://www.youtube.com/channel/UCxX9wt5FWQUAAz4UrysqK9A

Lectures and Annotations:

LESSON 1 – PRELIMINARIES OF ALGORITHM

An **Algorithm** for a particular task can be defined as *"a finite sequence of instructions, each of which have a clear meaning and can be performed with a finite amount of effort in a finite length of time"*.

Algorithm must be precise enough to be understood by human beings. However, in order to be executed by a computer, we will generally need a program that is written in a rigorous formal language; and since computers are quite inflexible compared to the human mind; programs usually need to contain more details than **Algorithms**.

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.*

A. 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.

B. CHARACTERISTICS OF AN ALGORITHM

Not all procedures can be called an Algorithm. An Algorithm should have the following characteristics:

- **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs / outputs should be clear and must lead to only one meaning.
- **Input** − An Algorithm should have 0 or more well-defined inputs.
- **Output** − An Algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** − Algorithms must terminate after a finite number of steps.
- **Feasibility** − Should be feasible with the available resources.
- **Independent** − An Algorithm should have step-by-step directions, which should be independent of any programming code.

How to Write an Algorithm?

We design an Algorithm to get a solution of a given problem. A problem can be solved in more than one ways.

There are no well – defined standards for writing Algorithms. Rather, it is problem and resource dependent. Algorithms are never written to support a particular programming code. As we know that all programming languages share basic code constructs like loops (*do, for, while*), flow – control (*if-else*), etc. These common constructs can be used to write an Algorithm.

We write Algorithms in a step-by-step manner, but it is not always the case. Algorithm writing is a process and is executed after the problem domain is well – defined. That is, we should know the problem domain, for which we are designing a solution.

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
```

In Design and Analysis of Algorithms, usually the second method is used to describe an Algorithm. It makes it easy for the analyst to analyze the Algorithm ignoring all unwanted definitions. He can observe what operations are being used and how the process is flowing.

Writing "Step Numbers", is optional. We design an Algorithm to get a solution of a given problem. A problem can be solved in more than one ways.

C. ALGORITHM ANALYSIS AND COMPLEXITIES

Algorithm Analysis

Efficiency of an Algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following:

- A **Priori Analysis** − This is a theoretical analysis of an Algorithm. Efficiency of an Algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.
- A **Posteriori Analysis** − This is an empirical analysis of an Algorithm. The selected Algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required are collected.

Algorithm Complexity

Annotation:

We can easily identify the efficiency of an Algorithm by means of measuring the running time and the memory space required from a single program.

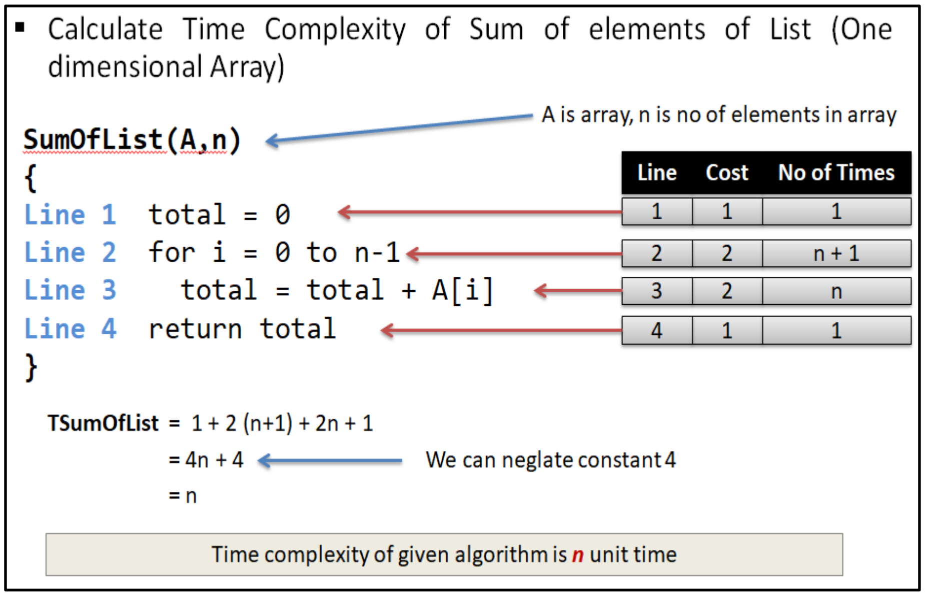
Suppose X is an Algorithm and n is the size of input data, the time and space used by the Algorithm X are the two main factors, which decide the efficiency of X.

- **Time Factor** – Time is measured by counting the number of key operations such as comparisons in the sorting and searching Algorithm.

- **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. Time Complexity



- Calculate Time Complexity of Sum of elements of List (One dimensional Array)

```
SumOfList(A,n)
{
Line 1  total = 0
Line 2  for i = 0 to n-1
Line 3    total = total + A[i]
Line 4  return total
}
```

A is array, n is no of elements in array

Line	Cost	No of Times
1	1	1
2	2	n + 1
3	2	n
4	1	1

TSumOfList = 1 + 2 (n+1) + 2n + 1
= 4n + 4 ← We can neglate constant 4
= n

Time complexity of given algorithm is ***n*** unit time

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 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.

2. 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:

- 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.

- 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)
Step 1 -  START
Step 2 -  C ← A + B + 10
Step 3 -  Stop
```

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

D. ASYMPTOTIC ANALYSIS AND NOTATIONS

Asymptotic Analysis

Asymptotic Analysis of an Algorithm refers to defining the mathematical bound / framing of its run-time performance. Using Asymptotic Analysis, we can very well conclude the *Best Case, Average Case, and Worst Case* scenario of an Algorithm:

- **Best Case Scenario** − Minimum Time required for program execution.
- **Average Case Scenario** − Average Time required for program execution.
- **Worst Case Scenario** − Maximum Time required for program execution.

Asymptotic Analysis is input bound i.e., if there's no input to the Algorithm, it is concluded to work in a constant time. Other than the "*input*" all other factors are considered constant. **Asymptotic Analysis** refers to computing the running time of any operation in mathematical units of computation. The running time of both operations will be nearly the same if *n* is significantly small. Usually, the time required by an algorithm falls under three types:

Asymptotic Notations

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

1. **Big Oh Notation, O**

   The Notation **O (n)** is the formal way to express the upper bound of an Algorithm's running time. It measures the *Worst Case Time Complexity* or the longest amount of time an Algorithm can possibly take to complete.

2. **Omega Notation, Ω**

   The Notation **Ω (n)** is the formal way to express the lower bound of an Algorithm's running time. It measures the *Best Case Time Complexity* or the best amount of time an Algorithm can possibly take to complete.

3. **Theta Notation, θ**

   The Notation **θ (n)** is the formal way to express both the lower bound and the upper bound of an Algorithm's running time.

E. RECURSION

The process in which a function calls itself directly or indirectly is called **Recursion** and the corresponding function is called a **Recursive Function.** Using a Recursive Algorithm, certain problems can be solved quite easily. Examples of such problems are:

- *Tower of Hanoi (TOH)*
- *In – Order / Pre – Order / Post – Order Tree Traversals*
- *DFS of Graphs*

Need of Recursion:

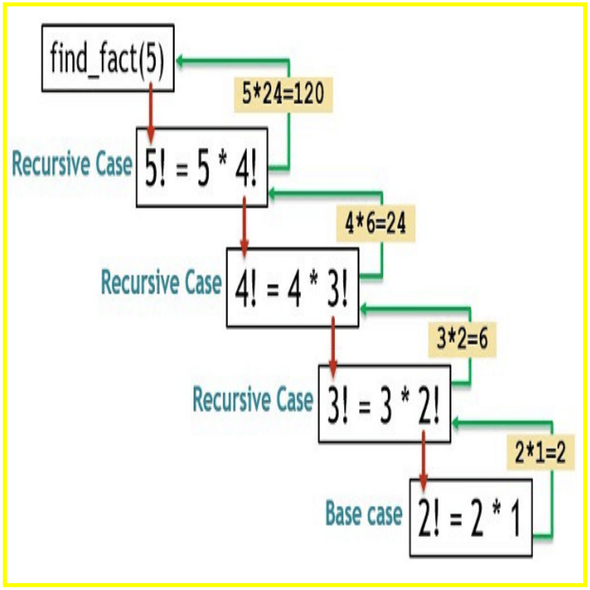
Recursion is an amazing technique with the help of which we can reduce the length of our code and make it easier to read and write. It has certain advantages over the iteration technique; for example, the *Factorial of a Number*.

Properties of Recursion:

- Performing the same operations multiple times with different inputs.

- In every step, we try smaller inputs to make the problem smaller.
- Base condition is needed to stop the Recursion otherwise infinite loop will occur.

Google wants me to do recursion, by clicking recursion, so that i can actually get the meaning of recursion theoretically and practically as well xd.



(https://www.geeksforgeeks.org/recursion/)

F. PSEUDOCODE AND FLOWCHART

Forms of Algorithms

1. **Pseudocode** – informal language which is basically a combination of the constructs of programming language, a narrative of the program language.

Example: 1, 2 – input 3 – sum (output)

```
Input = num1, num2
Sum = num1, num2

Display heading message
        "num1, num2, sum"
Display num1, num2, sum

End
```

2. **Flowchart** – defined as a pictorial representation describing a procedure or traditional graphic tool using standard symbols; a diagram that uses a set of symbol to show the sequence of steps in solving problem. To the programmer, a flowchart is a kind of an all – purpose tool; it is the "blueprint" of a program.

Frank Gilbert introduced Flowcharts in *1921*, and they were called "**Process Flow Charts**" at the beginning. **Allan H. Mogensen** is credited with training business people on how to use Flowcharts.

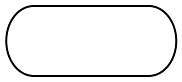
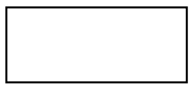
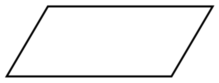
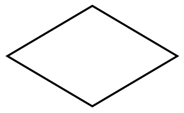
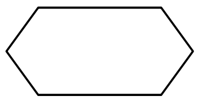
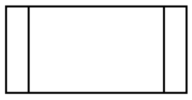
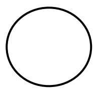
Types of Flowchart:

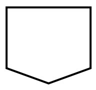
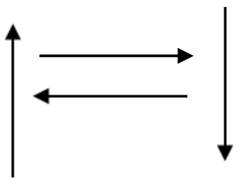
a. **Program Flowchart** – describing graphically in default the logical and steps within a program.

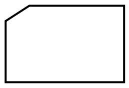
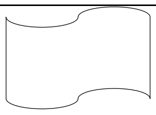
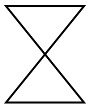
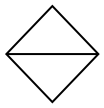
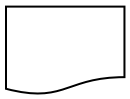
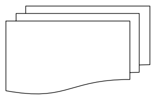
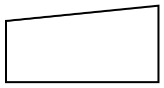
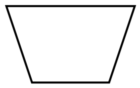
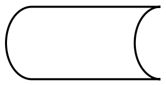
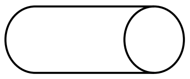
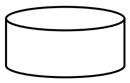
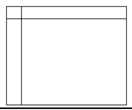
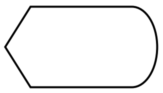
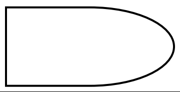
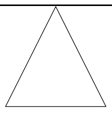
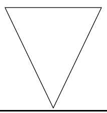
b. **System Flowchart** – graphically representations of the procedures involved in connecting data.

Flowchart Symbols

In this section, you will learn and identify the different shapes that commonly used when illustrating a flowchart with some of their uses and description.

	Flowchart Symbols:	**Flowchart Name:**	**Description:**
1.		*Terminal*	Used to designate the beginning and the end of a program, or a point of interruption.
2.		*Processing*	Used to represent a group of program instructions that perform a processing function of the program such as to perform arithmetic operations.
3.		*Input / Output Data*	Used to represent instructions to an input or an output device.
4.		*Decision*	Denotes a point in the program where more than one path can be taken or the decision making of a flowchart.
5.		*Preparation / Initialization*	It is commonly used to specify operations such as control, index register, initialization, switch setting, and in indicating loops.
6.		*Predefined Process*	As a subroutine symbol, it can be used when a procedure needs to be repeated several times.
7.		*On – Page Connector*	A non - processing symbol which is used to connect one part of a flowchart to another without drawing flow lines.

8.		*Off – Page Connector*	A type of connector used instead of the on- page connector to designate entry to or exit from a page when a flowchart requires more than one page.
9.		*Flow Line / Arrowhead*	Flow lines are used to use to show reading order or sequence in which flowchart symbols are to be read. Arrowheads are used to show the direction of processing or data flow.

10.		*Punch Card*	11.		*Punch Tape*
12.		*Collate*	13.		*Sort*
14.		*Document*	15.		*Multi - Document*
16.		*Manual Input*	17.		*Manual Operation*
18.		*On – Line Storage / Stored Data*	19.		*Magnetic Drum / Direct Access Storage*
20.		*Magnetic Disc*	21.		*Internal Storage*
22.		*Display*	23.		*Delay*
24.		*Extract*	25.		*Merge*

Steps in Problem Solving (Algorithm)

- First produce a general Algorithm (one can use Pseudocode)
- Refine the Algorithm successively to get step by step detailed algorithm that is very close to a computer language.
- Pseudocode is an artificial and informal language that helps programmers develops algorithms. Pseudocode is very similar to everyday English.

1. **Problem Solving Phase**
    - Produce an ordered sequence of steps that describe solution of problem
    - This sequence of steps is called an algorithm

2. **Implementation Phase**
    - Implement the program in some programming language.

Example*:*

Write an Algorithm to determine a student's final grade and indicate whether it is passing or failing. The final grade is calculated as the average of four marks.

Algorithm:

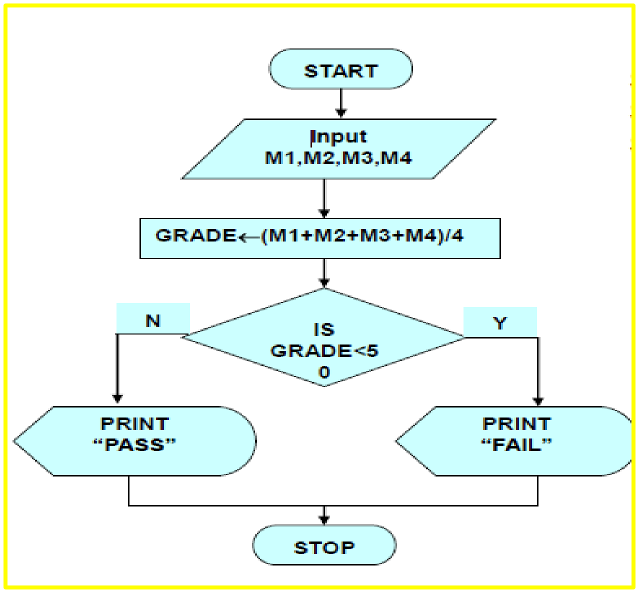
```
Step 1:     Input M1, M2, M3, M4
Step 2:     GRADE (M1 + M2 + M3 + M4) / 4
Step 3:     if (GRADE < 50) then
                Print "FAIL"
            else
                Print "PASS"
            Endif
```

Pseudocode:

```
Input a set of 4 marks.
Calculate their average by summing and dividing by 4.

if average is below 50
    Print "FAIL"
else
    Print "PASS"
```

Flowchart:



Operators Commonly Used in Flowcharting

A. **Arithmetic Operators**

+	-	Addition
-	-	Subtraction
*	-	Multiplication
/	-	Division
%	-	Modulus

B. **Relational Operators**

=	-	Equal
>	-	Greater Than
<	-	Less Than
<>	-	Not Equal
>=	-	Greater Than or Equal to
< =	-	Less Than or Equal to

C. **Logical Operators**

&&	-	Logical AND
||	-	Logical OR
!	-	Logical NOT

A	**B**	**A && B**	**A II B**
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
FALSE	TRUE	FALSE	TRUE
FALSE	FALSE	FALSE	FALSE

! TRUE	FALSE
! FALSE	TRUE

XDXDXDXDXDXDXDXDXDXDXDXDXDXDXD