**MODULE CODE AND NAME: SWD401: DATA STRUCTURE AND ALGORITHM FUNDAMENTALS**

**Competence: Apply Data Structure and Algorithm Fundamentals Using JavaScript**

**Learning outcome 1: Apply Algorithm Fundamentals Apply Algorithm Fundamentals Learning outcome 1: Apply Algorithm Fundamentals**

* 1. **Conversion of number systems**

1.1.1 Description of key concepts

**numeration system**

All number systems have the same three characteristics: **digits**, **base** and **weight**

**i. Decimal numeration system**

 The decimal numeration system uses only ten ciphers (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9) used in "Weighted" positions to represent very large and very small numbers.

 Each cipher represents an integer quantity, and each place from right to left in the notation represents a multiplying constant, or weight, for each integer quantity.

**ii. Binary numeration system**

1. The binary numeration system uses only two ciphers instead of ten as the decimal numeration system.

Those two ciphers are “0” and “1”. In binary system of numeration, ciphers are called bit (Binary Digit).

1. Cipher are arranged right to left in doubling values of weight (instead of multiplying the weight by 10 as in the case of decimal system).

**Remark:** With n bits we can represent **2*n*** different binary numbers. The higher H number is given using the following formula. **H =2n – 1**

**iii. Octal Number System**

The octal number system has a base or radix of eight, meaning that it has eight possible digits or symbols: 0, 1,2,3,4,5,6,7. How do you count in Octal number system?

1. **Hexadecimal Number System**

The hexadecimal system uses base 16. Thus, it has 16 possible digit symbols. It uses the digits

0, 1, 2, 3… 9 plus the letters A, B, C, D, E, and F.

How do you count in Hexadecimal number system?

1. **Unary encoding**

Unary coding, also known as the unary numeral system or thermometer code, is an entropy encoding that represents a natural number n with a code of length n+1 (or n), usually n ones followed by a zero (if natural number is understood as non-negative integer) or with n-1 ones followed by a zero (if natural number is understood as strictly positive integer).

Eg: 5=111110.

Unary coding has several applications in computer science and information theory. For example, it is used in the design of variable-length codes and Huffman coding

1.1.2 Number system from decimal base to:

**i. Decimal to binary conversion**

* In the most popular method called the dibble-dabble method, the given decimal number is successively divided by 2 giving a succession of remainders of **0** or
* The remainders read in the reverse order give the binary equivalent of the given decimal number.

1. **Decimal to octal conversion**

Decimal to octal conversion can be done by the dibble-dabble method by successively dividing by 8, giving succession of remainders lying between 0 and 7.

The remainders written in reverse order give the octal equivalent of given decimal number.

1. **Decimal to hexadecimal conversion**

Decimal to hexadecimal conversion can be done by the dibble-dabble method of successively dividing by 16 giving succession of remainders lying between 0 and 15. Then the remainders between 10 and 15 are renamed in hex as follows 10→A, 11→B, 12→C, 13→D, 14→E,

15→F.

The remainders written in reverse order give the hexadecimal equivalent of the given decimal number.

* 1. **Binary to Decimal conversion**

Convert 1011.1012 to decimal

1011.1012 = (1 x 23) + (0 x 22) + (1 x 21) + (1 x 20) + (1 x 2-1) + (0 x 2-2) + (1 x 2-3) =

(1 x 8) + (0 x 4) + (1 x 2) + (1 x 1) + (1 x 1/2) + (0 x 1/4) + (1 x 1/8) = 8 + 0 + 2 + 1 + 0.5 + 0 + 0.125 = 11.625

* 1. **Binary-To-Octal / Octal-To-Binary Conversion**
* To convert a binary number into octal simply divide the number into groups of three bits each starting at the binary point.
* Then express each group by its decimal (or octal) equivalent.
  1. **Binary-To-Hexadecimal / Hexadecimal-To-Binary Conversion**

**Binary-To Hexadecimal**

**Steps:**

1. Write down the full binary number
2. Split the number into 4 bit groups starting from the right.
3. Substitute the equivalent hexadecimal digit for each group.

People use the decimal number system to perform arithmetic operations. Computers, on the other hand, use the binary system, which contains only two digits: 0 and 1. We need a way to convert numbers from one system to another.

* **Application of number base arithmetic operations**

Number base arithmetic operations are used to perform arithmetic operations like addition, subtraction, multiplication, and division in various radix bases. These arithmetic operations may also use r’s complement and (r-1)’s complement representation.

For example, binary is the most commonly used non-base 10 system. It is used for coding in computers. Binary is also known as Base 2. This means it is composed of only 0’s and 1’s. Addition, subtraction, and multiplication in base 2 are performed similarly to the same way base 10 does, only smaller, therefore, requires more digits to make up the same number as in base 10.

In general, the base is defined as the total number of digits that are available in the number system. This is known as a positional number system. The most commonly used number system is the decimal system, commonly known as base 10. Its popularity as a system of counting is most likely due to the fact that we have 10 fingers

**1.2** **Description of logic gates and expressions**

* **Representation of Boolean logic gates**

**1.2.1 Boolean gates**

The table used to represent the Boolean expression of a logic gate function is commonly called a **Truth Table**. A logic gate truth table shows each possible input combination to the gate or circuit with the resultant output depending upon the combination of these input(s).

For example, consider a single **2-input** logic circuit with input variables labelled as A and B.

There are “four” possible input combinations or 22 of “OFF” and “ON” for the two inputs.

However, when dealing with Boolean expressions and especially logic gate truth tables, we do not general use “ON” or “OFF” but instead give them bit values which represent a logic level

“1” or a logic level “0” respectively.

Then the four possible combinations of A and B for a 2-input logic gate is given as:

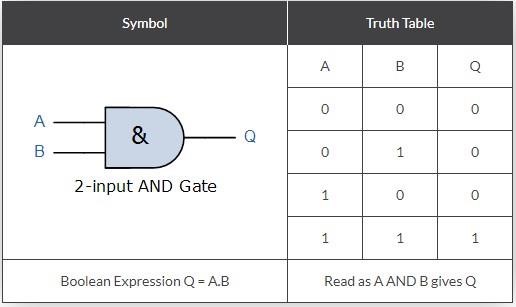
* Input Combination 1. – “OFF” – “OFF” or (0, 0)
* Input Combination 2. – “OFF” – “ON” or (0, 1)
* Input Combination 3. – “ON” – “OFF” or (1, 0)
* Input Combination 4. – “ON” – “ON” or (1, 1)

Therefore, a 3-input logic circuit would have 8 possible input combinations or 23 and a 4-input logic circuit would have 16 or 24, and so on as the number of inputs increases. Then a logic circuit with “n” number of inputs would have 2n possible input combinations of both “OFF” and “ON”.

So in order to keep things simple to understand, in this tutorial we will only deal with standard **2-input** type logic gates, but the principals are still the same for gates with more than two inputs.

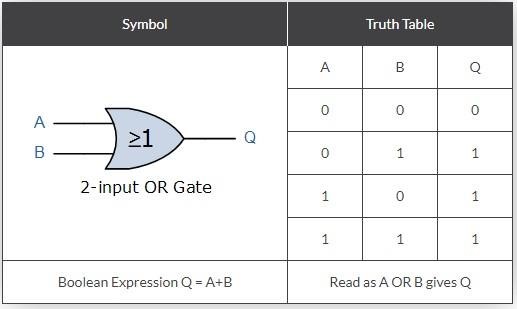
Then the Truth tables for a 2-input AND Gate, a 2-input OR Gate and a single input NOT Gate are given as: **2-input AND Gate**

For a 2-input AND gate, the output Q is true if BOTH input A “AND” input B are both true, giving the Boolean Expression of: (Q = A and B).



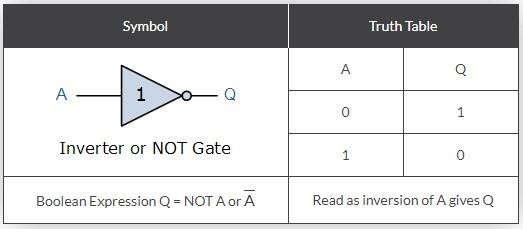
Note that the Boolean Expression for a two input AND gate can be written as: A.B or just simply AB without the decimal point. **2-input OR (Inclusive OR) Gate**

For a 2-input OR gate, the output Q is true if EITHER input A “OR” input B is true, giving the Boolean Expression of: (Q = A or B ).



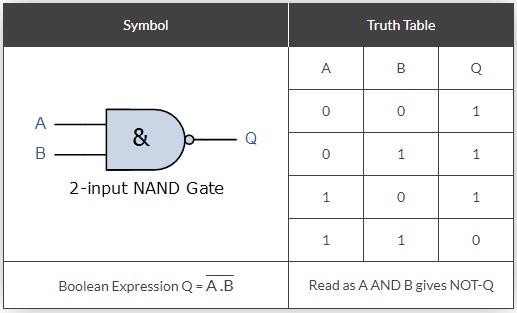
**NOT Gate (Inverter)**

For a single input NOT gate, the output Q is ONLY true when the input is “NOT” true, the output is the inverse or complement of the input giving the Boolean Expression of: (Q = NOT A).



The NAND and the NOR Gates are a combination of the AND and OR Gates respectively with that of a NOT Gate (inverter). **2-input NAND (Not AND) Gate**

For a 2-input NAND gate, the output Q is true if BOTH input A and input B are NOT true, giving the Boolean Expression of: (Q = not(A AND B) ).



**2-input NOR (Not OR) Gate**

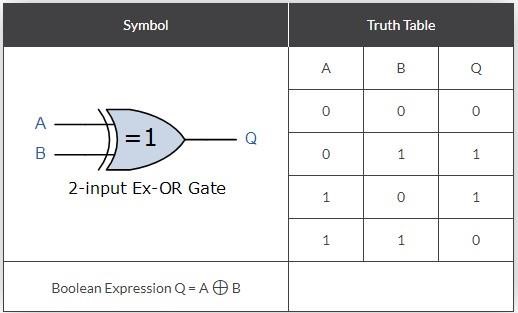
For a 2-input NOR gate, the output Q is true if BOTH input A and input B are NOT true, giving the Boolean Expression of: (Q = not (A OR B) ).

As well as the standard logic gates there are also two special types of logic gate function called an Exclusive OR Gate and an Exclusive-NOR Gate. The Boolean expression to indicate an

Exclusive-OR or Exclusive-NOR function is to a symbol with a plus sign inside a circle, (  ).

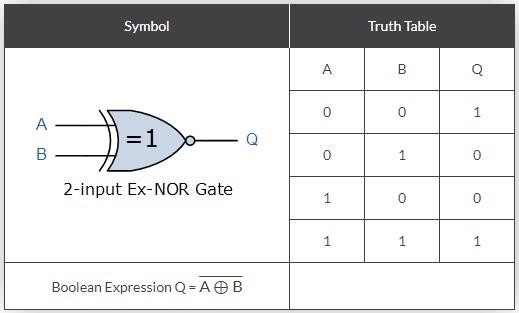
The switching actions of both of these types of gates can be created using the above standard logic gates. However, as they are widely used functions they are now available in standard IC form and have been included here as reference. **2-input EX-OR (Exclusive OR) Gate**

For a 2-input Ex-OR gate, the output Q is true if EITHER input A or if input B is true, but NOT both giving the Boolean Expression of: (Q = (A and NOT B) or (NOT A and B)).



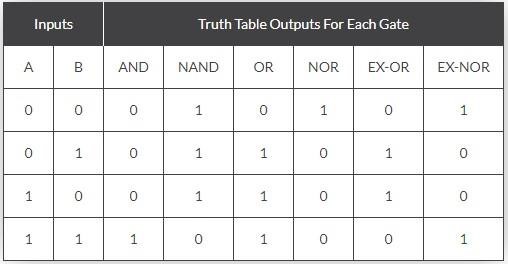
**2-input EX-NOR (Exclusive NOR) Gate**

For a 2-input Ex-NOR gate, the output Q is true if BOTH input A and input B are the same, either true or false, giving the Boolean Expression of: (Q = (A and B) or (NOT A and NOT B) ).

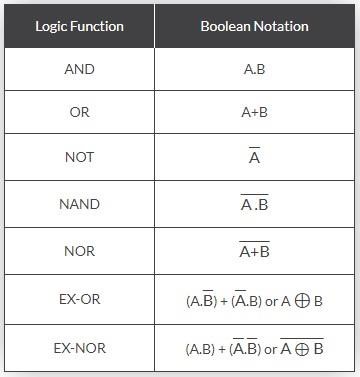


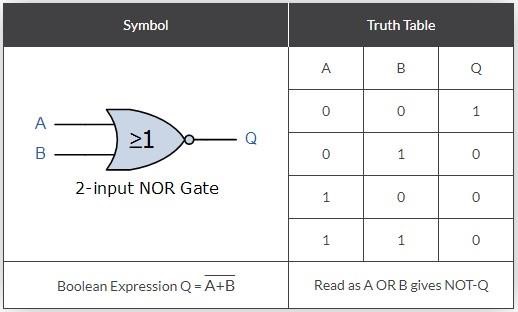
**Summary of 2-input Logic Gates**

The following Truth Table compares the logical functions of the 2-input logic gates above.



The following table gives a list of the common logic functions and their equivalent Boolean notation.





**Application of Boolean logic gates**

Boolean logic gates are digital devices that are mainly based on the Boolean function. They are used to carry out logical operations on single or multiple binary inputs and give one binary output. The most common Boolean operators are AND, OR, and NOT. Each operator has a standard symbol that can be used when drawing logic gate circuits.

Boolean logic gates have a wide range of applications in digital circuits. They are used in the architecture of our telephones, laptops, tablets, and memory devices. They are also used in search engines on the internet. In addition, they are used in the generation of parity generation and checking units

**1.3 Use of data types on variables**

Definition

Data types are used to define the type of data that a variable can hold in a program. They are used to ensure that the program is free from errors and runs efficiently.

Datatype: determines what kind of data is being stored in a variable.

There are two types of data types in JavaScript namely Primitive data types and Non-primitive data types.

**1.Primitive data types:**The predefined data types provided by JavaScript language are known as primitive data types.

Primitive data types are also known as in-built data types.

**Eg:**

* String: A sequence of characters enclosed in single or double quotes.
* Number: A numeric value, which can be an integer or a floating-point number.
* BigInt: A numeric value with arbitrary precision.
* Boolean: A logical value that can be either true or false.
* Undefined: A variable that has been declared but not assigned a value.
* Null: A variable that has been explicitly assigned the value null.
* Symbol: A unique and immutable primitive value that can be used as the key of an object property

2.**Non-primitive data types:** The data types that are derived from primitive data types of the JavaScript language are known as non-primitive data types. It is also known as derived data types or reference data types.

Eg: Array, Object etc.

* 1. **Application of JavaScript operators**

JavaScript operators are used to perform specific computations on operands. They are used for comparing values, performing arithmetic operations, and more. Here is a brief overview of the different types of JavaScript operators you mentioned:

**1. Arithmetic operators**

+ Addition

- Subtraction

\* Multiplication

/ Division

↑or ̂Power

**2. Comparison operators (**Relational operators)

These operators are used to compare two values and return a Boolean value (true or false). The most common relational operators are:

> Greater than

< Less than

>= Greater than or equal to

<= Less than or equal to

== Equal to

! = Not equal to

**3. Logic operators**

AND operator (&&)

OR operator (||)

and NOT (!).

**4.** **Assignment operator**

To put a value in a variable we use an assignment operator which has the following symbol: ←

These operators are used to assign values to variables. The most common assignment operator is the equals sign (=)

1. **Compound operators:**

These operators combine an arithmetic operator with an assignment operator. For example, the += operator adds a value to a variable and assigns the result back to the variable.

1. **Conditional operators:**

These operators are also known as ternary operators because they take three operands. They are used to assign a value to a variable based on a condition. The most common conditional operator is the ternary operator (? :)

1. **Bitwise operators:**

These operators are used to perform bitwise operations on binary numbers. The most common bitwise operators are AND (&), OR (|), XOR (^), NOT (~), left shift (<<), and right shift (>>)

* 1. **Algorithm**

**How to Write an Algorithm?**

**What is an Algorithm?**

**1.5.1 Definition**

* Algorithm is the sequence of instructions which are involved in solving a given

problem or accomplishment of a given task.

* It can be translated into a programming language in order to produce the result.
* Algorithm is a method of representing the step-by-step logical procedure for solving a

problem.

* Algorithms can be represented in two main ways: pseudocode and flowcharts.
* Therefore, Algorithm refers to a set of rules/instructions that step-by-step define how a work is to be executed upon in order to get the expected results.
* Algorithm is a well-defined sequential computational technique that accepts a value or a collection of values as input and produces the output(s) needed to solve a problem.
* An algorithm is a step-by-step procedure to solve a problem or perform a task. An algorithm is independent of the language used. It tells the programmer the logic used to solve the problem.

**1.5.2 There are several reasons why we use algorithms:**

• Efficiency: Algorithms can perform tasks quickly and accurately, making them an essential tool for tasks that require a lot of calculations or data processing.

• Consistency: Algorithms are repeatable and produce consistent results every time they are executed. This is important when dealing with large amounts of data or complex processes.

• Scalability: Algorithms can be scaled up to handle large datasets or complex problems, which makes them useful for applications that require processing large volumes of data.

• Automation: Algorithms can automate repetitive tasks, reducing the need for human intervention and freeing up time for other tasks.

* Standardization: Algorithms can be standardized and shared among different teams or organizations, making it easier for people to collaborate and share knowledge.

**1.5.3 Types of algorithm**

* 1. **Sorting algorithms**: Bubble Sort, insertion sort, and many more. These algorithms are used to sort the data in a particular format.
  2. **Searching algorithms**: Linear search, binary search, etc. These algorithms are used in finding a value or record that the user demands.

**3. Graph Algorithms**: It is used to find solutions to problems like finding the shortest path between cities, and real-life problems like traveling salesman problems.

**4. Recursive Algorithm**

This is one of the most interesting Algorithms as it calls itself with a smaller value as inputs which it gets after solving for the current inputs. In simpler words, It’s an Algorithm that calls itself repeatedly until the problem is solved.

**5. Divide and Conquer Algorithm**

This is another effective way of solving many problems. In Divide and Conquer algorithms, divide the algorithm into two parts; the first parts divide the problem on hand into smaller sub problems of the same type. Then, in the second part, these smaller problems are solved and then added together (combined) to produce the problem’s final solution.

**6. Dynamic Programming Algorithm**

These algorithms work by remembering the results of the past run and using them to find new results. In other words, a dynamic programming algorithm solves complex problems by breaking them into multiple simple sub problems and then it solves each of them once and then stores them for future use.

**7. Greedy Algorithm**

These algorithms are used for solving optimization problems. In this algorithm, we find a locally optimum solution (without any regard for any consequence in future) and hope to find the optimal solution at the global level.

The method does not guarantee that we will be able to find an optimal solution.

**8. Brute Force Algorithm**

This is one of the simplest algorithms in the concept. A brute force algorithm blindly iterates all possible solutions to search one or more than one solution that may solve a function. Think of brute force as using all possible combinations of numbers to open a safe.

**9. Backtracking Algorithm**

Backtracking is a technique to find a solution to a problem in an incremental approach. It solves problems recursively and tries to solve a problem by solving one piece of the problem at a time. If one of the solutions fail, we remove it and backtrack to find another solution.

**1.5.4 Description of qualities of a good algorithm**

**a. Finiteness**: the algorithm stops after a finite number of instructions are executed.

**b. Definiteness**: Each step of an algorithm must be precisely defined; the actions to be

carried out must be rigorously and unambiguously specified for each case.

**c. Input**: the algorithm receives input. An algorithm has zero or more inputs, i.e,

quantities which are given to it initially before the algorithm begins.

**d. Output**: the algorithm produces output. An algorithm has one or more outputs i.e,

quantities which have a specified relation to the inputs.

**e. Effectiveness:** An algorithm is also generally expected to be effective. This means that

all of the operations to be performed in the algorithm must be sufficiently basic that they can

in principle be done exactly and in a finite length of time.

**f. Precision** – the steps are precisely stated (defined).

**g. Uniqueness** – results of each step are uniquely defined and only depend on the input

and the result of the preceding steps.

**h. Generality** – the algorithm applies to a set of inputs.

**I**. **Independent:** An algorithm must be language independent, which means that it should mainly focus on the input and the procedure required to derive the output instead of depending upon the language.

**Advantages of Algorithms**

**Easy to understand:** Since it is a stepwise representation of a solution to a given problem, it is easy to understand.

**Language Independent:** It is not dependent on any programming language, so it can easily be understood by anyone.

**Debug / Error Finding:** Every step is independent / in a flow so it will be easy to spot and fix the error.

**Sub-Problems:** It is written in a flow so now the programmer can divide the tasks which makes them easier to code.

**Disadvantages of Algorithms**

• Creating efficient algorithms is time-consuming and requires good logical skills.

• Nasty(unpleasant) to show branching and looping in algorithms

**1.5.5 Develop an algorithm using structured English**

**1. Sequence structures**

An algorithm can be written in English like sentences or in any standard representation

Sometimes, algorithm written in English like languages are called Pseudo Code.

Example:

Suppose we want to find the average of three numbers, the algorithm

is as follows

Step 1 Read the numbers a, b, c

Step 2 Compute the sum of a, b and c

Step 3 Divide the sum by 3

Step 4 Store the result in variable d

Step 5 Print the value of d

Step 6 End of the program

Example:

Var A as Integer

Start

A=5

End

Explanations:

Var A as Integer is the variable declaration line

Start marks the beginning of an algorithm

A=5 the instructions part

End marks the end of an algorithm

**1.5.5.1Creating an Algorithm:**

Since the algorithm is language-independent, we write the steps to demonstrate the logic behind the solution to be used for solving a problem. But before writing an algorithm, keep the following points in mind:

• The algorithm should be clear and unambiguous.

• There should be 0 or more well-defined inputs in an algorithm.

• An algorithm must produce one or more well-defined outputs that are equivalent to the desired output. After a specific number of steps, algorithms must ground to a halt.

• Algorithms must stop or end after a finite number of steps.

• In an algorithm, step-by-step instructions should be supplied, and they should be independent of any computer code.

**1.5.5.2 Design Algorithm**

An algorithm is made mainly of the following parts:

The variable declaration line

The beginning of an algorithm

The instructions part

The end

Example: algorithm to multiply 2 numbers and print the result:

Step 1: Start

Step 2: Get the knowledge of input. Here we need 3 variables; a and b will be the user input and c will hold the result.

Step 3: Declare a, b, c variables.

Step 4: Take input for a and b variable from the user.

Step 5: Know the problem and find the solution using operators, data structures and logic

We need to multiply a and b variables so we use \* operator and assign the result to c.

That is c = a \* b

Step 6: Check how to give output, here we need to print the output. So write print c

Step 7: End

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

Follow the algorithm approach as below:

Step 1: Start the Program

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 of Solution

**1.5.5.3 Variables**

A variable is a memory zone which is used to store data. It is characterized by a name, an

address and a data type.

* Variable name: a variable name helps to distinguish it from other variables.
* Variable address: helps to locate it in the memory.
* Variable data type: helps to know the operations allowed to be performed on it and

the size it occupies in the memory.

**Variable declaration**

A variable to be used must first be declared. Declare a variable; means create it by giving it a

name and a data type.

**Rules for giving a valid name to a variable**

* + Start a variable name with an upper case letter
  + Don’t use a blank space in variable name
  + Instead of using a blank space in a variable name put underscore
  + Don’t use reserved words
  + Don’t use special characters

1. **Selection/conditional structures**

Whereas in branch control, there is a condition and according to a condition, a decision of

either TRUE or FALSE is achieved. In the case of TRUE, one of the two branches is explored;

but in the case of FALSE condition, the other alternative is taken. Generally, the ‘IF-THEN’ is used to represent branch control.

**1. if**

IF condition THEN

if body

ENDIF

Example: Write an algorithm to find the smallest number between two numbers

Step1: Start

Step2: Input two numbers, say A and B

Step3: If A<B then small = A

Step4: If B<A then Small = B

Step5: Print Small

Step 6: End

Example 2: Write an algorithm to check odd or even number

Step1: Start

Step2: Read/Input a number

and store in A Step3: Is A<0?

If YES, then C=” ODD”

If NO, then c=” even”

Step4: Display C

Step5: Stop

Example 3: Write an algorithm to find the maximum of all the elements present in the array.

Follow the algorithm approach as below:

Step 1: Start the Program

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 of Solution

* + 1. **Develop an algorithm using pseudocode**

Pseudo code is an informal high-level description of the operating principle of a computer program or other algorithm

An algorithm should be seen as a starting point when writing a program. The finished program should follow the steps the algorithm describes.

Before an algorithm can be designed, it is important to check that the problem is completely decomposed. The decomposed problem should consider:

* the inputs into the problem
* the outputs of the problem
* the order in which instructions need to be carried out
* any decisions that need to be made in the problem
* any areas of the problem that are repeated

Only when a problem is properly decomposed and understood can an algorithm design begin.

Programs are developed using programming languages. These languages have specific syntax that must be used so that the program will run properly.

Pseudocode is not an actual programming language. Instead, it is a simple way of describing a set of instructions in a manner that resembles a programming language. It has its own syntax, some of which is very similar to many actual programming languages.

Any algorithms designed using pseudocode will not run unless they are converted into an actual programming language.

**Special Keyword**

1. **START/BEGIN:**To begin the pseudocode.
2. **INPUT**: Take input from the user.
3. **PRINT:**To print the output on the screen.
4. **READ/GET:**Input format while reading data from the file.
5. **SET, INIT:**Initialize a value.
6. **INCREMENT, BUMP:**Increase the value of the variable, equivalent to a++.
7. **DECREMENT:**Decrease the value of the variable, equivalent to a--.
8. **COMPUTE, CALCULATE, DETERMINE:**To calculate the expression result.

**1.5.6.1 Read and write Keyword**

**Read Keyword (Inputs)**

A read **Keyword** is a Keyword which is used for inputs. It helps to receive the value entered by a user and assign it to a variable.

Syntax of read **Keyword**:

Read

Example:

Write an algorithm which receives a number entered by a user.

Answer:

Var A as Integer

Start

Read A

End

**Print Keyword**

Print **Keyword** is used for Outputs; it displays the content of a variable of displays messages.

Syntax of write **Keyword**:

Print “ ”

Example:

Write an algorithm which displays a value stored in a variable.

Answer:

Var B as Integer

Start

B=5

Print “The content of the variable is:”

Print B

End

**1.5.6.2 Tests**

There are situations in which a set of instructions are executed in one situation and entirely another set of instructions to be executed in a different situation. In this kind of situations, a decision

control instruction (test) is used. We can define a test as a structure which controls the flow of

instructions of a program during its execution. We can also define it as a structure, which helps us to evaluate a condition.

**1. Structure of a test**

The structure of a test is made of two main parts: the part which evaluate a condition, and a part of one instruction or a block of instructions.

**2. Conditions (if, if…else, switch)**

**2.1. If statement**

Syntax

IF condition THEN

if body

ENDIF

The if statement is used to make a decision. The block of instructions following the if executes if the decision is true, and the block does not execute otherwise.

Example 1:

start

If the light is green then

Go

End if

End

Example 2:

start

If the light is red then

Stop

End if

End

Each of these statements is conditional. If the condition is true, the instruction following the

condition go in the first example and stop in the second example are executed. In case the condition

evaluates to false nothing is done.

Example 3:

Write an algorithm which receives a number and informs the user when it is positive.

Answer:

Var A as integer

Start

Print “enter a number”

Read A

If A>0 then

Print “the number is positive”

End if

End

To this question when the condition evaluates true it displays the instruction: the number is positive but when it evaluates for false it displays nothing.

**2.2. If…else statement**

Syntax

IF condition THEN

if body

ELSE

else body

ENDIF

The if…else statement is used to make a decision and gives the alternative when the condition

evaluates to false. The block of instructions following the if executes if the decision is true, and

the block after else when the condition evaluates to false.

Example 1:

start

If the light is green, then

Go

else

Stop

End if

End

If the condition is true, the instruction following the condition go is executed, when it evaluates to

false the instruction stop which follows else is executes.

Example 2:

Write an algorithm which receives a number and informs the user whether it is positive or negative.

Answer:

Var A as integer

Start

INPUT: “enter a number”

Read A

If A>0 then

Print “the number is positive”

else

Print “the number is negative”

End if

End

To this question when the condition evaluates true it displays the message the number is positive

when it evaluates to false it displays the message the number is negative.

For eg:

age = INPUT: "Enter your age"

IF age is greater than 18

PRINT "adult"

ELSE

PRINT "Under age"

ENDIF

END

**2.3. Nested Ifs (if…else if…else)**

If statement may be used inside another if statement, in such case we call it a nested if.

IF condition statement THEN

if body

ELSE IF condition THEN

else if statement

ELSE

else body

ENDIF

For eg:

age = INPUT: "Enter Your age"

IF age is equal to 18 THEN

PRINT "under check"

ELSE IF age is greater than 18

PRINT "Give entry"

ELSE

PRINT "under age"

ENDIF

END

Example:

Write an algorithm which receives student note and it displays the grade as follows:

Note form 16 and above: Grade A

Note 14-16 : Grade B

Note 12-14 : Grade C

Note below 12 : Grade D

Answer:

Var Note as integer

Start

Print “enter the note”

Read Note

If Note>=16 then

Print “Grade A”

Else if Note>=14 then

Print “Grade B”

Else if Note>=12 then

Print “Grade C”

Else

Print “Grade D”

End if

end

**2.4 Multiple choice using ‘switch’**

A multiple choice using switch helps to solve the problem caused by nested if statement in case

there are many conditions to be tested. Switch receives a variable then it evaluates it using several

Case.

Syntax:

Menu explaining to the user how to make a choice

Switch(variable)

Case 1

Instruction

Case 2

Instruction

….

Case n

Instruction

Default

Instruction

End switch

Example:

Write an algorithm which receives note and displays the student’s grade.

16 and above: grade A

14-16: grade B

12-14: grade C

Less than 12: grade D

Answer:

Var Note as integer

Start

Print “Use number to choose the range of your note”

Print “enter 1 for note ranging from 16 and above”

Print “enter 2 for note ranging from 14 to16”

Print “enter 3 for note ranging from 12 to14”

Print “enter 4 for note less than 12”

INPUT: “enter your choice now using number:”

Read(Note)

Switch(Note)

Case 1

Print “You have grade A”

Case 2

Print “You have grade B”

Case 3

Print “You have grade C”

Case 4

Print “You have grade D”

Default

Print “Your choice is not listed, try again”

End switch

End

**3.1 Loops**

**3.1**.1. What is a loop?

A loop helps to repeat instruction or block of instructions. It assists in the algorithm where you want to carry out an activity for a certain number of times.

**3.1.2. Do while loop**

The do-While loop execute its statements at least once even if the condition fails for the first time. It tests the condition at the end of loop body.

Syntax

Variable=<start value>

Do while <variable><comparison operator><end value>

Instruction or block of instructions

Variable=variable+1

Loop

Example:

Write an algorithm which use do while loop and displays numbers from 1 to 10

Answer:

Var A as integer

Start

A=1

Do while A<=10

Write(A)

A=A+1

Loop

End

**3.1.3. Do… loop Until**

Do Until Loop means to do something until the condition becomes TRUE. It is like a logical

function that works based on TRUE or FALSE. This is the opposite of the Do While loop where

Do while runs the loops as long as the condition is TRUE.

Syntax:

<Variable>=<start value>

Do

Instruction or block of instructions

Variable=variable+1

Loop until while <variable><comparison operator><end value>

Example:

Write an algorithm which use do loop until and displays numbers from 1 to 10

Answer:

Var A as integer

Start

A=1

Do

Print A

A=A+1

Loop until A>10

End

**3.1.4. For loop**

The for loop is an iterative loop, it specifies some elements about the loop in one single line.

a. Setting a loop counter to an initial value.

b. End which determine whether its value has reached the number of repetitions desired

The value of the loop counter will be increased each time (iteration), and segment within the loop

will be executed.

Syntax:

FOR initial\_value TO end\_value

for body

ENDFOR

EXAMPLE:

FOR i -> 0 to 20

PRINT i

ENDFOR

example:

Write an algorithm which ask a user to enter a number and it displays the 10 next numbers.

Var I, A as integer

Start

Print “enter a number”

Read I

For A=1 to 10 do

I<-I+1

Print I

End for

End

**3.1.5. Loops in Loops**

Loops in loops refer to what we call nested loops. These are loops that are such that when one

increment by one the other continues inside the first.

Var I, J as Integer

start

For I = 1 To 9 do

For J = 1 To I do

Write(J)

End for

End for

End

**3.1.6 while loop**

WHILE condition

while body

ENDWHILE

Example:

i : 0

WHILE i <= 2

PRINT i

ENDWHILE

**3.4 Handle a table (Array)**

**3.4.1 Use of tables in algorithm (Arrays)**

An array is a variable with elements of the same data type. To access the array elements, we use the array index.

For example, if you need to record notes of 20 students in a given course, you will have to declare 20 variables as follows:

Var N1, N2, N3, N4, N5, N6, N7, N8, N9, N10, N11, N12, N13, N14, N15, N16, N17, N18, N19, N20 as integer

Writing these variables and assigning values to them is hard.We can use one variable called an array to hold all these numbers.

**3.4.2 Declaring an array**

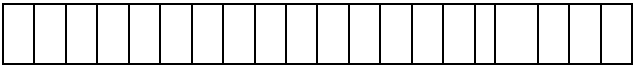
Syntax

Var <array name> <size> as <data type>

Example:

Var N(19) as integer

This array will look as follows:



Variable i is used in a loop to move from index 0 of the array to index 19 performing given

instructions on the array

**3.4.3 How do we assign values to an array?**

Let us declare an array which will hold five numbers and assign to it 5 numbers of our choice.

Var N(4) as integer

Start

N(0) =8

N(1) = 10

N(2) =12

N(3) = 6

N(4) =5

End

To display the content of this array we will need a loop.

Var i as integer

For i=0 to 4 do

Print N(i)

End for

If we put together these parts we will have

Var N(4) as integer

Var i as integer

Start

N(0)=8

N(1) =10

N(2) =12

N(3) =6

N(4) =5

For i=0 to 4 do

Print N(i)

End for

End

In case we are told that the user we enter numbers in array we will use another array for input, and our algorithm will change as follows:

Var N(4) as integer

Var I as integer

Start

For I=0 to 4 do

read N(I)

End for

For I=0 to 4 do

Print N(I)

End for

End

**3.4.4 Pseudocode**

* It can be understood as one of the methods that helps in the representation of an algorithm.
* It is a simpler version of coding in a programming language.
* It is written in plain English, and uses short phrases to write the functionalities that specific line of code would do.
* There is no specific syntax which is actually present in other programming languages.
* This means it can't be executed on a computer.
* There are many formats that could be used to write pseudo-codes.
* Pseudocode is not actually a programming language.
* Control structures such as 'while', 'if-then-else', 'repeat-until', and so on can be used.

**Example of an Algorithm and Pseudocode that add two numbers entered by the User**

**Algorithm**

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

**Pseudocode**

BEGIN

NUMBER s1, s2, sum

OUTPUT ("Input number1:")

INPUT s1

OUTPUT ("Input number2:")

INPUT s2 sum=s1+s2

OUTPUT sum

END

Example of an Algorithm and Pseudocode for the greatest number between three

numbers

 Algorithm

STEP 1: START

STEP 2: Declare three variables X, Y, Z

STEP 3: If X>Y&X>Z print X is the GREATEST

STEP 4: Otherwise if Y>Z&Y>X print Y is the GREATEST

STEP 5: Otherwise print Z is the GREATEST

STEP 6: STOP

 Pseudocode

BEGIN

NUMBER X, Y, Z

INPUT X

INPUT Y

INPUT Z

IF X>Y AND X>Z THEN

OUTPUT X + " is the GREATEST"

ELSE IF Y > X AND Y>Z THEN

OUTPUT Y + " is the GREATEST"

ELSE

OUTPUT Z + " is the GREATEST"

ENDIF

END

Exercises and solutions

For eg: **1. Guess an integer in a range**

**Algorithm:**

Step1: Start

Step 2: Declare hidden, guess

Step 3: Compute hidden= Choose a random value in a range

Step 4: Read guess

Step 5: If guess=hidden, then

Print Guess is hit

Else

Print Guess not hit

Print hidden

Step 6: Stop

**Pseudocode:**

BEGIN

COMPUTE hidden=random value in range

READ guess

IF guess=hidden, then

PRINT Guess is hit

ELSE

PRINT Guess not hit

PRINT hidden

END IF-ELSE

END

**2. Find minimum in a list**

**Algorithm:**

Step 1: Start

Step 2: Read n

Step 3: Initialize i=0

Step 4: If i<n, then goto

step 4.1, 4.2 else goto

step 5 Step4.1: Read a[i]

Step 4.2: i=i+1 goto step 4

Step 5: Compute min=a[0]

Step 6: Initialize i=1

Step 7: If i<n, then go to step 8 else goto step 10

Step 8: If a[i]<min, then goto

step 8.1,8.2 else goto 8.2

Step 8.1: min=a[i]

Step 8.2: i=i+1 goto 7

Step 9: Print min

Step 10: Stop

**Pseudocode:**

BEGIN

READ n

FOR i=0 to n, then

READ a[i]

INCREMENT i

END FOR

COMPUTE min=a[0]

FOR i=1 to n, then

IF a[i]<min, then

CALCULATE min=a[i]

INCREMENT i

ELSE

INCREMENT i

END IF-ELSE

END FOR

PRINT min

END

**3. Insert a card in a list of sorted cards**

**Algorithm:**

Step 1: Start

Step 2: Read n

Step 3: Initialize i=0

Step 4: If i<n, then goto step 4.1, 4.2 else goto step 5

Step4.1: Read a[i]

Step 4.2: i=i+1 goto step 4

Step 5: Read item

Step 6: Calculate i=n-1

Step 7: If i>=0 and item<a[i], then go to step 7.1, 7.2 else goto step 8

Step 7.1: a[i+1]=a[i]

Step 7.2: i=i-1 goto step 7

Step 8: Compute a[i+1]=item

Step 9: Compute n=n+1

Step 10: If i<n, then goto step 10.1, 10.2  lse goto st 11

Step10.1: Print a[i]

Step10.2: i=i+1 goto step 10

Step 11: Stop

**Pseudocode:**

BEGIN

READ n

FOR i=0 to n, then

READ a[i]

INCREMENT i

END FOR

READ item

FOR i=n-1 to 0 and item<a[i], then

CALCULATE a[i+1]=a[i]

DECREMENT i

END FOR

COMPUTE a[i+1]=a[i]

COMPUTE n=n+1

FOR i=0 to n, then

PRINT a[i]

INCREMENT i

END FOR

END

**1.5 Write Flow chart**

* A flowchart is a type of diagram that represents an algorithm or process, showing the

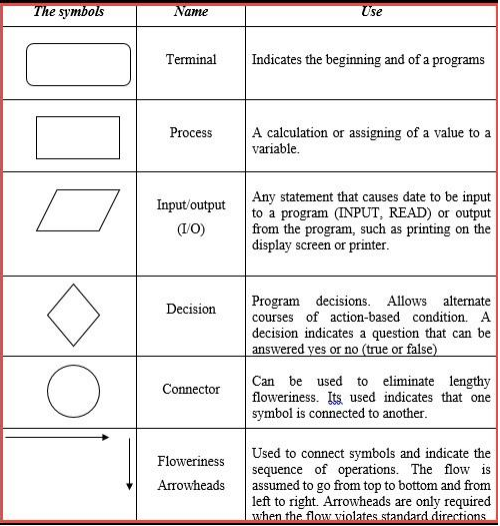
steps as boxes of various kinds, and their order by connecting these with arrows.

* A flowchart is a is pictorial representation of step by step solution of a problem.
* A flowchart is the graphical representation of an algorithm with the help of different symbols, shapes and arrows in order to demonstrate a process or a program. With algorithms,

we can easily understand a program.

* Programmers use it as a program-planning tool in order to solve a problem

The main purpose of a flowchart is to analyze different processes. Several standard graphics are applied in a flowchart:



**1.5.1 How to Use Flowcharts to Represent Algorithms?**

Now that we have the definitions of algorithm and flowchart, how do we use a flowchart to

represent an algorithm?

Algorithms are mainly used for mathematical and computer programs, whilst flowcharts can

be used to describe all sorts of processes: business, educational, personal and of course

algorithms. So flowcharts are often used as a program planning tool to visually organize the

step-by-step process of a program. Here are some examples:

Example 1: Print 1 to 20:

Algorithm:

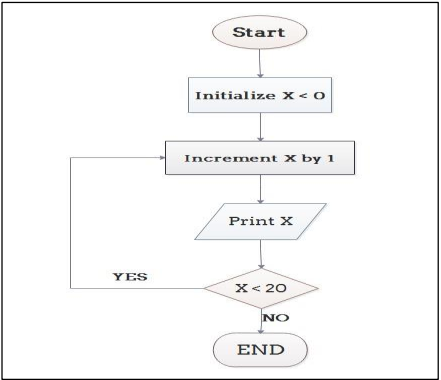
Step 1: Initialize X as 0,

Step 2: Increment X by 1,

Step 3: Print X,

Step 4: If X is less than 20 then go back to step 2.

Flowchart:



Example 2: Convert Temperature from Fahrenheit (℉) to Celsius (℃)

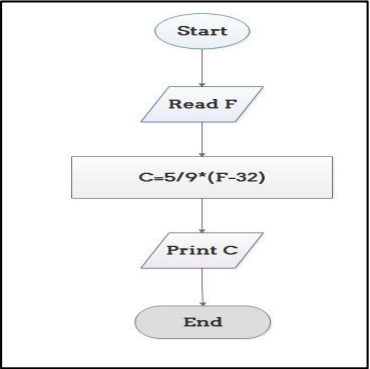
Algorithm:

Step 1: Read temperature in Fahrenheit,

Step 2: Calculate temperature with formula C=5/9\*(F-32),

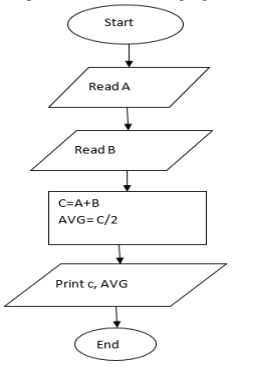
Step 3: Print C,

Flowchart:

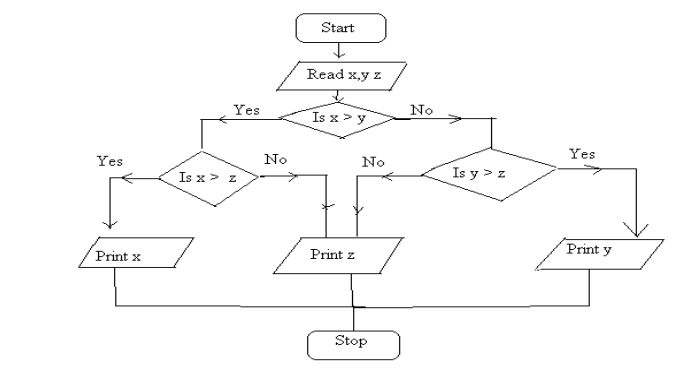


Example 2: A Flow chart for program that adds two numbers and calculate the average

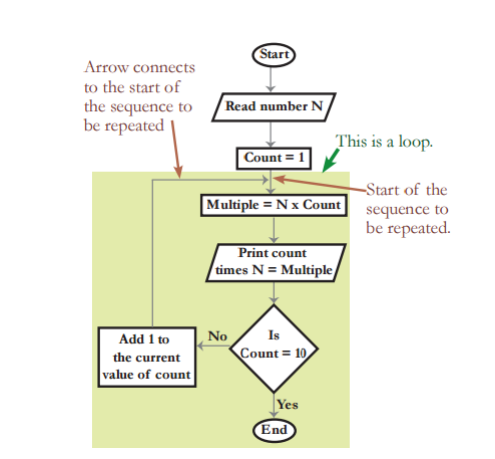
Flowchart:



Example 3: A Flow chart for program that calculate the largest number between three (3) numbers

Flowchart: 

Eg 4: A flowchart for program that display the Multiplication table for a given number N



Task: Q1. Mr. HABUMUGISHA has a work to find all the roots of a quadratic equation ax2+bx+c=0 1. Draw a flowchart to find all the roots of a quadratic.

Q2. What do you understand by flowchart and list all the standard graphics applied in a flowchart?

Q3. Write an algorithm to find the average of the marks of 5 subjects inputted from the

keyboard and display the display the following:

If Average<=100 and >=50 the program should display “PASS!!!!!!!!!!!”

If Average<50 the program should display “FAIL!!!!!!!!!!!!!”

Q4. Differentiate algorithm and pseudocode?

END OF LO1