

Introduction of the track of Algorithm and Data structures,

Welcome to our algorithms and data structure course !

This course is for computer science student or any one who want to learn basics of algorithms. It also helps for job interviews.

Big companies ask data structures and algorithms questions to see if you know how to think like programmer and how to design fast and scalable algorithms.

We will not use any program language because the purpose is to focus about logic, not on syntax.

Before we start, let's talk about the necessity of the algorithms.

Algorithms and logical thinking is a cornerstone in software engineering, so to be a better problem solver and developer, the first thing to do is to learn algorithmic thinking.

One more important note, during this course we are not going to concentrate on the programming languages or the syntax of any language, but the most important part is to learn how to logically think to make an optimized and scalable algorithms

During this chapter, we are going to learn:

What's the structure of an algorithm?

What are the variables and their identifiers?

What are types of variables?

What is sequential processing?

What is selection processing?

What is iterative processing?

Structure of an algorithm

What is an algorithm :

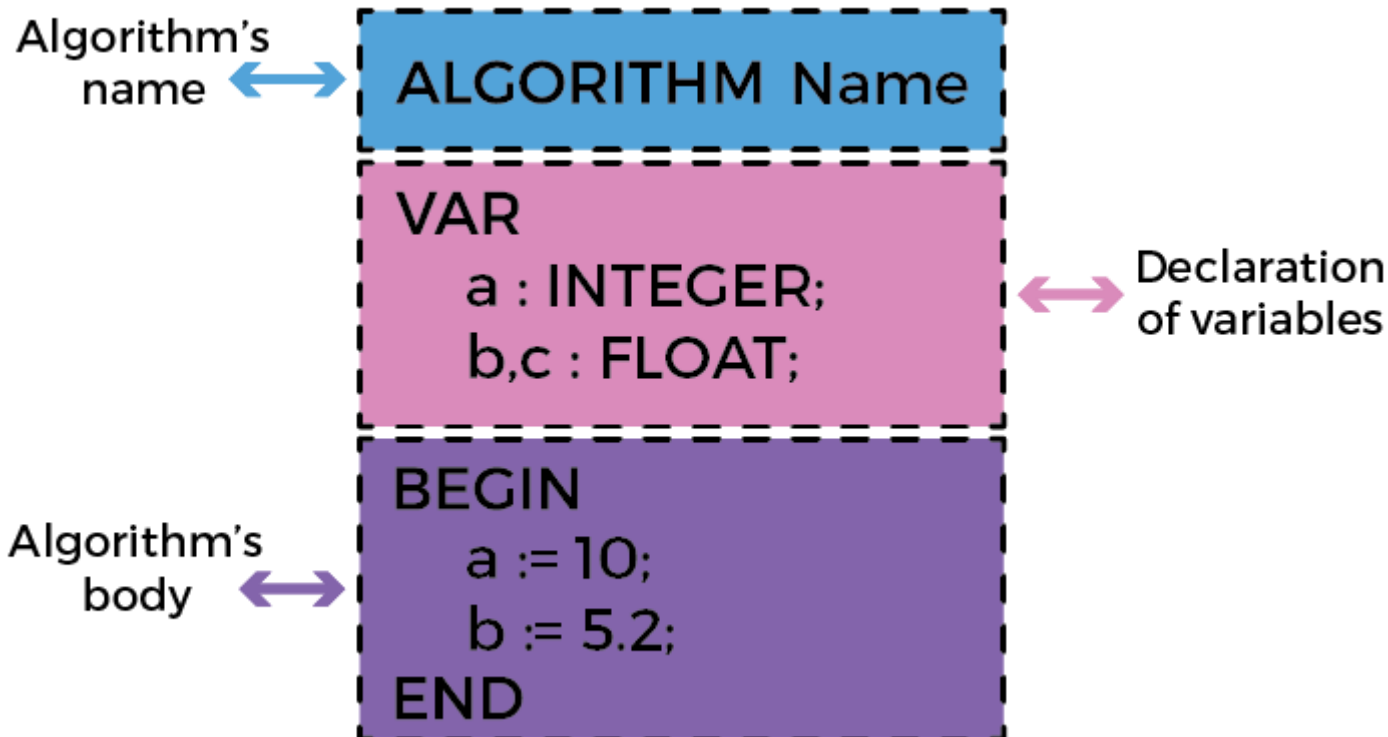
An algorithm is a set of instructions for solving a problem, in other words it's

a way of providing a result from data.

In other words, Algorithm presents a process or set of rules to be followed to obtain the expected output from the given input.



Structure of Algorithm :



Why do we write algorithms :

- Develop the logic of programming.
- Stay focused on the logic only.
- Check the complexity of the program

How to Design an Algorithm?

Before we start writing our algorithm some pre-requests must be fulfilled, in order to make sure that we are on the right path.

The followings are the information that we should gather before we start writing the algorithm

Define the problem that needed to be solved by the algorithm.

Identify the constraints of the problem that must be considered while solving the problem.

Gather the input token to solve the problem.

Recognize the output to be expected when the problem has solved.

Describe the solution to this problem, in the given constraints

Let's consider the following example, we need to add three numbers and print their sum. If we apply what we have learned from before we'll find something like this:

The problem: Add three numbers and print their sum.

The constraints: The numbers must contain only digits and no other characters.

The input: The three numbers to be added.

The output: the sum of the three numbers.

The solution: The solution consists of reading the three numbers, and adding them. it can be done using the '+' operator, or bit-wise, or any other method.

Now after gathering these pieces of information, we can start designing the algorithm.

Start //here we declare the beginning of our algorithm

Declare 3 integer variables num1, num2 and num3

Read the value of each variable // respectively put them into the variable as inputs

Declare an integer variable sum // this variable will contain the resultant sum

Add the three numbers and store the result in the variable sum

Print the value of the variable sum

END // here we declare the end of our algorithm

How to design an Algorithm?

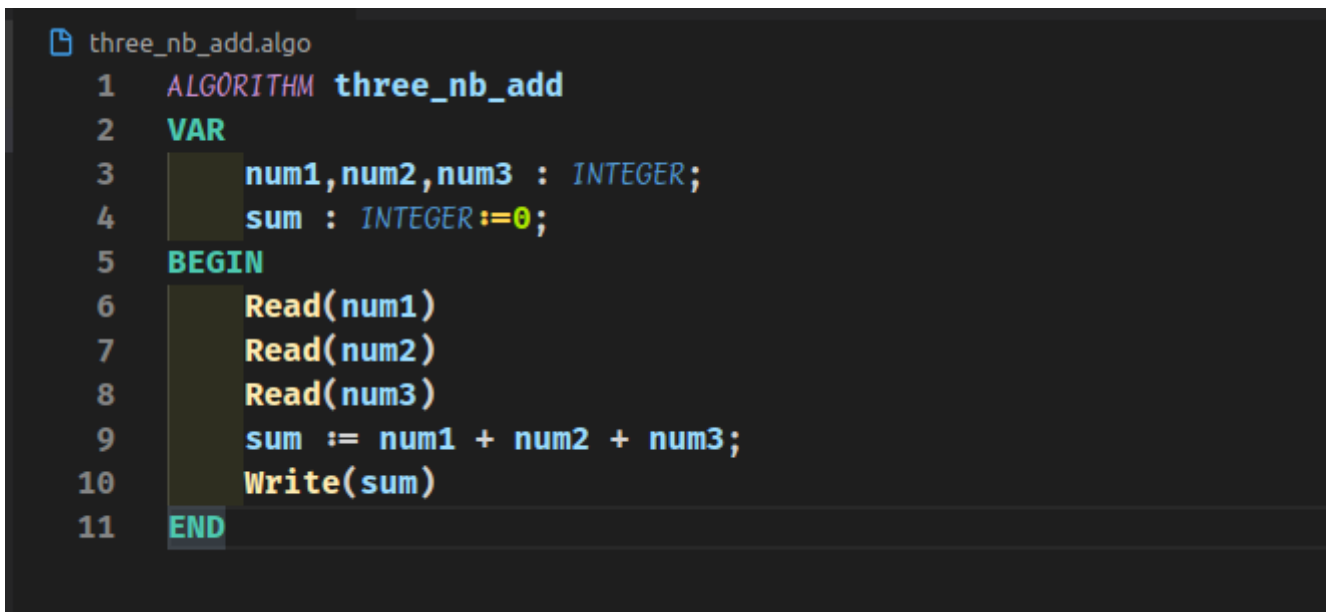
After defining the pre-requests and make a draft of our algorithm let's make it look like a real algorithm. To do so, we'll use the visual code as our default IDE and we have to add these two extensions to make it easier:

1. algo-gmc by "Hmida Rojabani": [Link](#)
2. indent-rainbow by "oderwat": [Link](#)

Now after setting up the environment, we need to create a file named three_nb_add with the extension ".algo"

The image below is the implementation of our algorithm in VSCode.

Go ahead and try it yourself !



```
three_nb_add.algo
1  ALGORITHM three_nb_add
2  VAR
3      num1, num2, num3 : INTEGER;
4      sum : INTEGER := 0;
5  BEGIN
6      Read(num1)
7      Read(num2)
8      Read(num3)
9      sum := num1 + num2 + num3;
10     Write(sum)
11  END
```

An algorithm is:

A programming language.

A set of steps of how to resolve a certain problem.

Variables and identifiers

What is a variable?

- Variables are used to associate a name with a value, which can change during the algorithm .
- In other words, a variable is a storage location for a single type of data, which can be used to store a value and used to retrieve a value.
- Variable declaration :

Variable declaration can be done in two ways :

1. Name_of_variable : type
2. Name_of_variable, name_of_other_varibale : type

```
age : integer // declare a variable with the name "age" of
type "integer"

a,b : boolean // declare two variables with the name "a" and
"b" of type "boolean"
```

To declare a constant (A variable that will not change its value throughout the algorithm) we use the keyword CONST instead of VAR

Identifier

- Variable name or identifier has precise rules for creation.
- Luckily, the same rules for identifiers apply to anything you are free to name, including variables, program name, procedures, functions and structures (we will see all of them in this course).
- There are only three rules to remember for legal identifiers:
 1. The name must begin with a letter (uppercase or lowercase) or the underscore symbol ' _ '.
 2. Subsequent characters may also be numbers.
 3. You cannot use the same name as an algorithm reserved word, such as var, begin ,end...

The following examples are legal:

- okidentifier
- OK2Identifier
- _alsoOK1d3ntifi3r
- __SStillOkbutKnotsonice

These examples are not legal:

- 3DPointClass // identifiers cannot begin with a number
- hollywood@vine // @ is not a letter, digit or _
- *_coffee // * is not a letter, digit or _

- `var` // `var` is a reserved word



Assignment :

Assignment a value to a variable can be done with four way :

- `Name_of_variable := value`
- `Name_of_variable := name_of_other_varibale`
- `Name_of_variable := expression`
- `Name_of_variable := name_of_function`

```
age := 28      // assign (i.e. store) the value 28 in the
variable "age"

b := false     // assign (i.e. store) the value false in the
variable "b"

a := b         // assign (i.e. store) the value of "b" (false)
in the variable "a"
```

Which of the following are valid identifiers?

A-B

_helloWorld

Program

1990_s

abc

Select the correct form of assignment to an integer variable a.

a=5

a==5

a:=5

a :: 5

Types of variables:

What are the variables types ?

Type	Utility	Example
Boolean	Represents a binary state, true or false.	true , false
Integer	Represents number that can be written without a fractional component.	26 , 305 , 7 , 4018
Float	Represents number that can be written with a fractional component.	12.14 , 25.0 , 3.1415 , 6
Char	Represents a unique character.	'c' , 'A' , '\$'
String	Represents a series of characters (text).	"content of the channel"

Compatibility of types

When you assign a value to a variable, the two types must be compatible with each other. For example, you cannot assign a string value to an integer variable.

Integer data type



Compatibility of types in operations

By now, we have learned what's are variables and the basics types that exist. As we have seen in the previous slide we cannot assign a value with type float to a variable with type string, this rule still true between variable themselves.

Only the exception of assigning an integer variable to a float variable.

As shown in the code below, we encourage you to try it yourself.

```
ALGORITHM types
```

```
VAR
```

```
  i : INTEGER := 26;
```

```
  f : FLOAT := 4.56;
```

```
  c : CHAR := 'c';
```

```

s : STRING := 'str';

b : BOOLEAN := TRUE;

BEGIN

    // this is a single line comment

    /* this is
    a multi line comment */

    f := i; // correct

    i := f; // wrong

    // we cannot mix types

    f := c; // wrong

    s := c; // correct

    b := i; // wrong

END

```

Let a an integer value and b a float value, which expression is correct?

a := a+b

b := a+b

What is the correct way to make a comment in an algorithm ?

// this is a comment

\\ this is a comment

/* this is a comment */

/\$ this is a comment \$/

Sequential processing

Executing a sequence of instructions

By now, we have learned how an algorithm is structured. The next step is to learn how to build an algorithm and what are its categories processing. As mentioned before, an algorithm is a plan or step-by-step instructions to solve a problem. There are three basic building blocks (constructs) to use when designing an algorithm: sequencing, selection, iteration

For now, we'll be interested in the sequencing pattern. Like its name describe the sequencing pattern is based on giving the computer the right sequence of instruction to follow.

Let's take an example of an algorithm designed to draw a square in 07 steps :

draw a 3 cm line

turn left 90 degrees

draw a 3 cm line

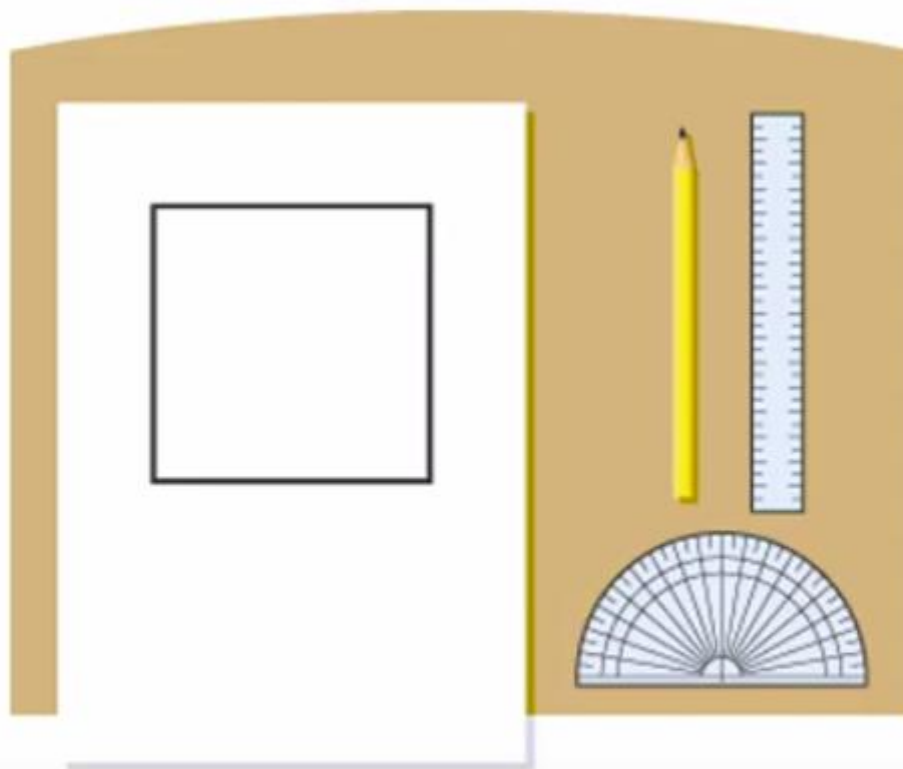
turn left 90 degrees

draw a 3 cm line

turn left 90 degrees

draw a 3 cm line

Following this algorithm, the result will be:



Now let's reverse the last two instructions, the algorithm will look like this:

draw a 3 cm line

turn left 90 degrees

draw a 3 cm line

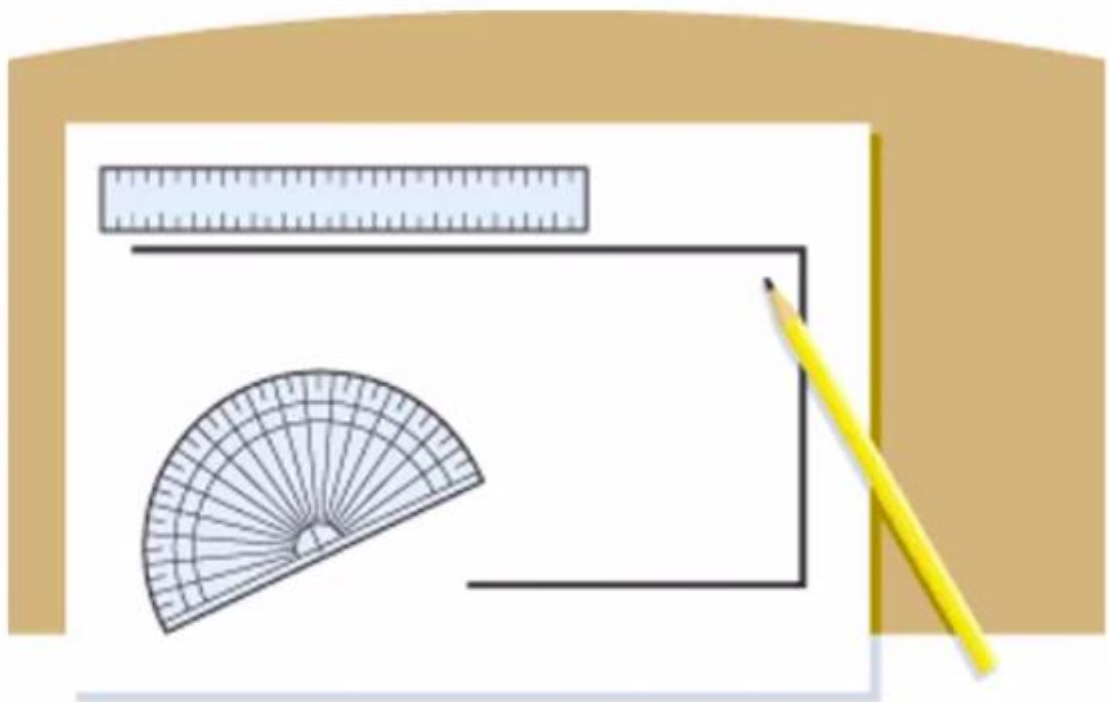
turn left 90 degrees

draw a 3 cm line

draw a 3 cm line

turn left 90 degrees

Well then this would be the result of the execution:



In reality, the algorithm will propose a set of instructions for the computer to do, in sequence (with a specific order) to solve the problem. However, a computer can execute only two categories of instructions:

Arithmetic operations: addition, multiplication, etc ...

Logical operations: in this type of instructions the computer can execute another two sub-categories which are:

Boolean: AND, OR, NOT, etc ...

Relational: =,>,<=, etc...

Arithmetic instructions

Arithmetic instruction or operation is like in mathematics, it is a manipulation of constants or variables.

Arithmetic operation	Example
Addition	outcome := a + b
Subtraction	outcome := a - b
Multiplication	outcome := a * b
Division	outcome := a / b
Modulo (division remainder)	outcome := a % b

Logical instructions

These instructions perform logical operations on data, such as comparing, complement, and, or...

Logical Operation	Example
Not	outcome := not a

Logical Operation		Example
Or		outcome := a or b
And		outcome := a and b
Equals		outcome := a = b
Not Equal		outcome := a <> b
Greater than / Greater than or equals		outcome := a > b /outcome := a >= b
Less than	Less than or equals	outcome := a < b /outcome := a <= b

What is the outcome of this operation : $5 \leq 4$

True

False

What is the type of this operation : $5 \leq 4$

Arithmetic

Relational

Boolean

Is this instruction correct : Not 10 OR TRUE

Correct

Not Correct

Selection processing

Algorithms

consist of a set of instructions that are carried out (performed) one after another. Sometimes there may be more than one path (or set of steps) that can be followed. At this point, a decision needs to be made. This point is known as selection. Depending on the answer given, the algorithm will follow certain steps and ignore others.

Why is selection important?

Selection allows us to include more than one path through an algorithm. selection is usually represented by the instructions **IF**, **THEN** and **ELSE**.

- IF represents the question
- THEN points to what to do if the answer to the question is true
- ELSE points to what to do if the answer to the question is false

create a conditional control:

When you have to create a conditional control in your program with more than 3 or 4 conditions, it becomes complicated with if- else, here, a SWITCH statement became a quite useful.

The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression. However, the switch statement can replace an If-Else statement in the case where we compare a variable to several integral values. You can use the switch statement only with the equals operation.

```
SWITCH (n) DO

    CASE 1 : // code to be executed if n = 1;

                BREAK; // BREAK is used to skip the other cases

    CASE 2 : // code to be executed if n = 2;

    DEFAULT : // code to be executed if n doesn't match any
cases
```

END_SWITCH

Selection processing in practice

In this slide, we are going to see a real example using the conditional statement, the IF-ELSE, and the Switch.

Let's start with the IF statement.

Our problem to solve is that the ticket price is variable according to the passenger age. if the age is under 16 then the ticket price will be half of the real price.

If the age over 16 then the ticket price will be the real price.

The statement will be like below:

```
/* ***** Bus tickets ***** */

/*
ask how old are you
IF you are under 16, THEN pay half ticket
ELSE pay full ticket
*/

// first solution
ticket_price := 20 ;
IF (age <=16) THEN
    ticket_price := 10 ;
END_IF

// Second solution
IF (age <= 16) THEN
    ticket_price := 10 ;
ELSE
    ticket_price := 20 ;
END_IF
```

Well let's try to make it a little harder, we'll have three test cases:

a case where the age is under 10, the passenger will pay only 20% of the real price;
a case where the age is between 10 and 16, the passenger will pay 50% of the ticket price.

a case where the age is above 16, the passenger will pay a full ticket.

The solution will look like:

```
IF (age<=10) THEN

    ticket_price := 4 ;

ELSE_IF (age<=16) THEN

    /* in this case, we don't specify whether the age is not
    under 10

    because in this case it will satisfy the first condition
    and execute the first block of instruction */

    ticket_price := 10 ;

ELSE

    ticket_price := 20 ;

END_IF
```

Selection processing in practice

Now let's try to resolve another problem using the switch statement.

Our problem is, we are going to print the opening hours of a zoo. The problem here is that the zoo opens at a different time according to the season. Example if it is summer the door get opened at 10:00 and close at 20:00

if it is winter it open at 10:00 and closed at 16:00

Now since we only have four seasons, we are going to represent these season as shown below

1 refer to winter

2 refer to spring

3 refer to summer

4 refer to autumn

And here is the solution to this problem:

```
/* **** Zoo time **** */

/*

1 refer to winter

2 refer to spring

3 refer to summer
```

```

4 refer to autumn
*/
SWITCH (season) DO
    case 1 : Write("10h00 to 16h00"); BREAK;
    case 2 : Write("10h00 to 18h00"); BREAK;
    case 3 : Write("10h00 to 20h00"); BREAK;
    case 4 : Write("10h00 to 16h00"); BREAK;
    default : Write("Wrong number") // optional use of BREAK;
END_SWITCH

```

Now the thing that we should notice in this code is the existence of the BREAK keyword. Let's suppose that the season is 3, and we remove the BREAK keyword when we execute the program display "10h00 to 20h00" "10h00 to 16h00" "Wrong number"

That's due to how the switch works if in any case, the condition is true then it will execute all the rest of the cases. and to make it right we add the BREAK keyword.

4 / 5

Is this code correct, explain your answer.

```

IF (n+2) THEN
    Write(n);
END_IF

```

Correct

Incorrect

Let s:=0, n:=1; What is the result of s after this code

```

SWITCH (n) DO
    CASE 1 : s:=s+1;
    CASE 2 : s:=s+1; BREAK;
    DEFAULT : s:=10;
END_SWITCH

```

s=0

s=1

s=2

s=3

Let n:= 3; What is the output of this code

```
SWITCH (n) DO
  DEFAULT : Write("Go");
  CASE 1 : Write("My");
  CASE 2 : Write("Code");
END_SWITCH
```

GoMyCode

Go My Code

Go My

Iterative processing

Iteration

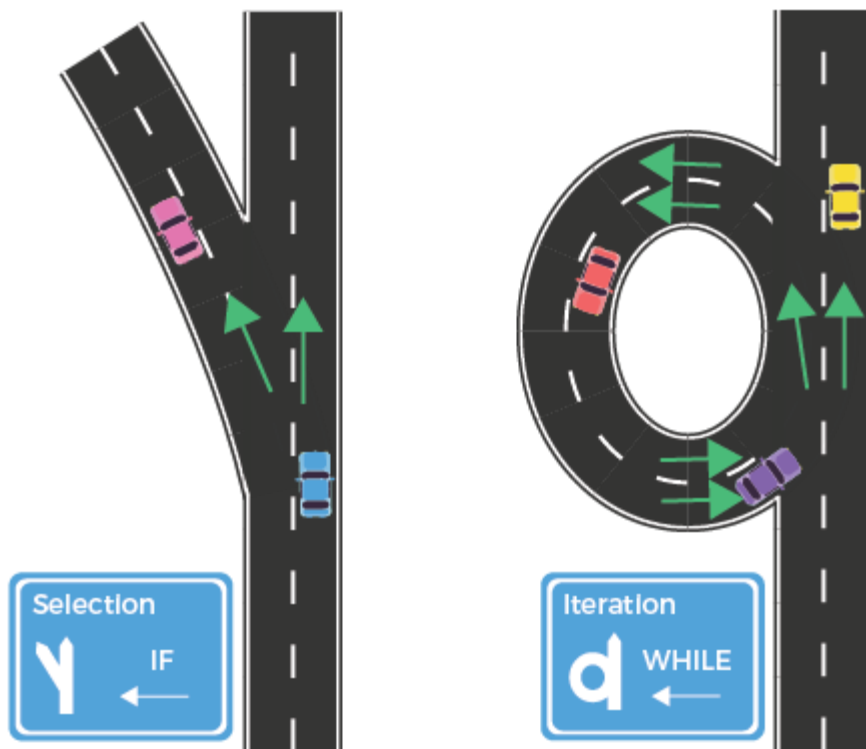
Iteration in programming means **repeating** steps, or instructions, over and over again. This is often called a **'loop'**.

Algorithms consist of instructions that are carried out (performed) one after another.

Sometimes an algorithm needs to repeat certain steps until told to stop or until a particular condition has been met.

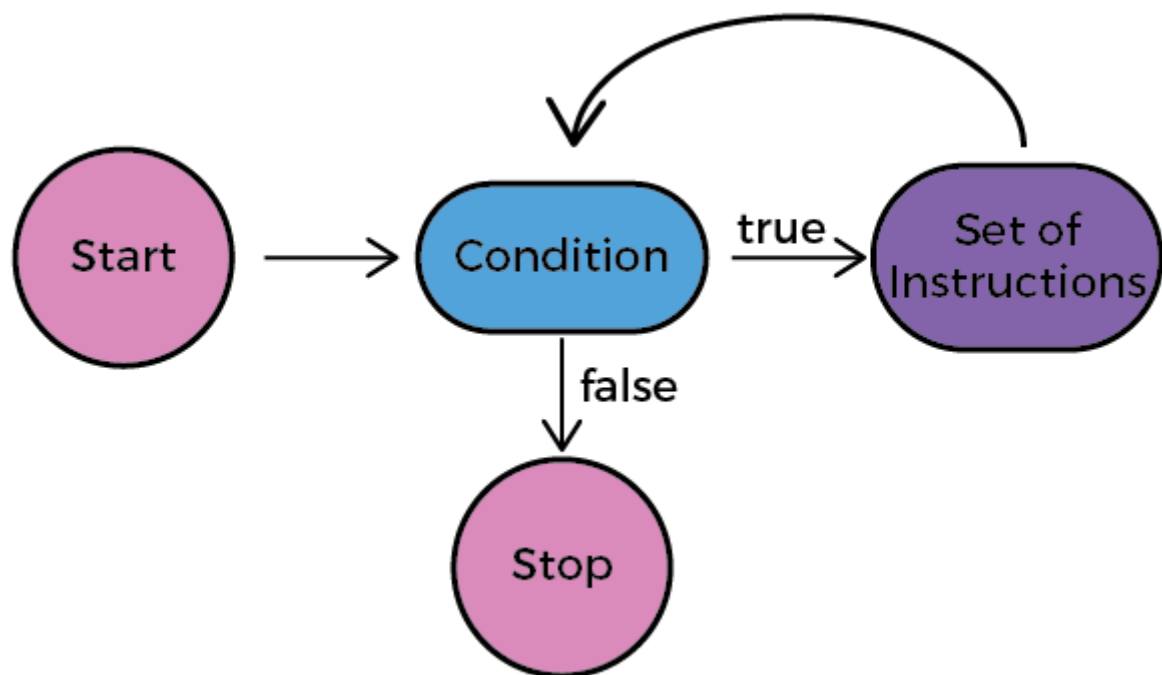
Iteration allows algorithms to be simplified by stating that certain steps will repeat until told otherwise. This makes designing algorithms quicker and simpler because they don't need to include lots of unnecessary steps.

Using condition with iteration is very similar to selection as two routes are created. However, with iteration there is a loop back into the algorithm, rather than creating and keeping two separate routes as would occur with selection.

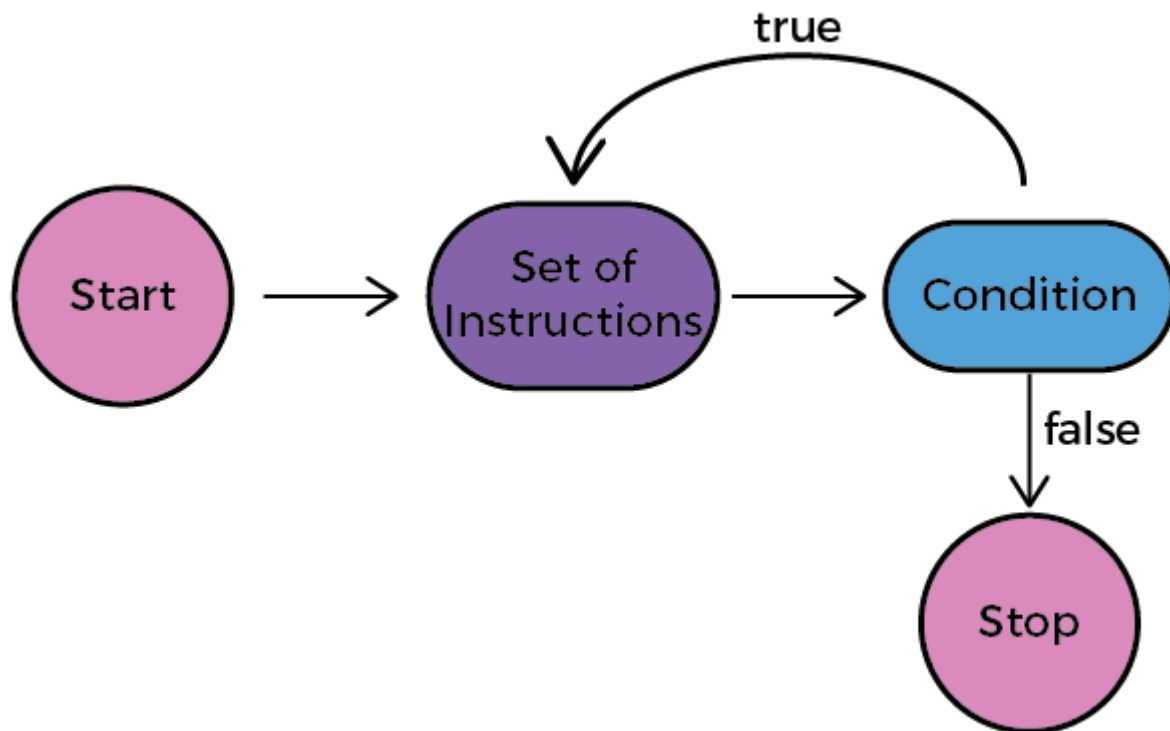


Iteration

A WHILE-DO loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The while loop can be thought of as a repeating if statement.



A REPEAT-UNTIL loop is similar to while loop with the only difference that it checks for the condition after executing the statements, and therefore is an example of Exit Control Loop. Here, the loop body will execute at least once.



FOR-DO loop

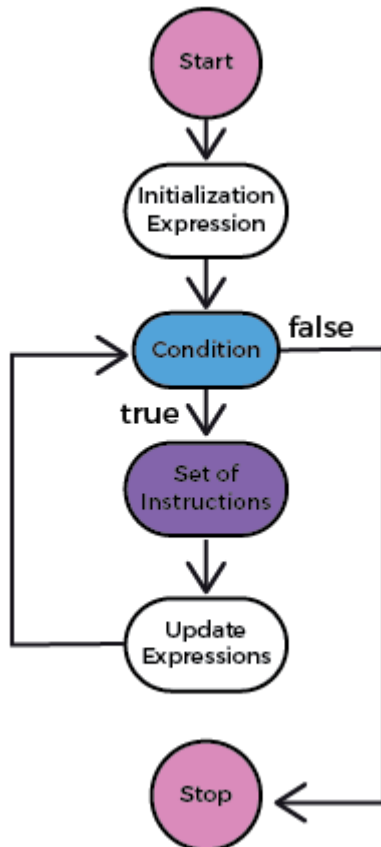
is a repetition control structure which allows us to write a loop that is executed a specific number of times. The loop enables us to perform n number of steps together in one line.

Unlike a while loop, a for statement consumes the initialization, condition and increment/decrement in one line thereby providing a shorter, easy to debug structure of looping.

How does a For loop executes?

- Control falls into the for loop. Initialization is done
 - The flow jumps to Condition
 - Condition is tested.
1. If Condition yields true, the flow goes into the Body
 2. If Condition yields false, the flow goes outside the loop
- The statements inside the body of the loop get executed.
 - The flow goes to the Updating (increment/decrement current value)
 - Updating takes place and the flow goes to Step 3 again

- The for loop has ended and the flow has gone outside.



Iterative processing in practice

Let's take an example to better understand the iterative processing.

In this type of processing, we have three kinds of iterators. We are going to cover them in these examples.

The first iterator is the for loop.

This kind of iterator is used when already knew how many times we are going to repeat the same instruction.

Let's say that we want to calculate the sum of the number from 1 to 10, in this example we already know how many time we are going to repeat the sum instruction.

The solution is shown below:

```

/* *** Calculate the Sum *** */

/* In this example we are going
to calculate the sum of number from
1 to 10 */
ALGORITHM iterative_for

```

```

VAR
i : INTEGER;
sum : INTEGER := 0;

BEGIN

FOR i FROM 1 TO 10 STEP step DO
/*i represents the iterator variable,
the 1 represents the starting point
10 the end point
step represents the step of the iteration and it's optional
if we don't put then the default step will be 1 */
    sum := sum + i
END_FOR

    Write(sum)

END

```

The second iterator is the while loop.

This iterator is used when we don't have any idea about the number of iterations.

Let's take this example, suppose that the user introduces a number and we have to return how many times this number can be divided by 2.

Here is the solution:

Every time we divide the number n by 2, we increment the count by 1 and we repeat these instructions until the remainder of the division by 2 is different than 0.

```

ALGORITHM iterative_while

VAR

count : INTEGER := 0;

// n is the number to test

n : INTEGER;

```

```

BEGIN

WHILE (n%2 = 0 ) DO

  /* while the condition is true
  then it will repeat the instruction
  until the condition will be false
  */

  n := n/2; // every time we divide the number n by 2 and
  count := count +1; // increment the count by 1

END_WHILE

Write(count)

END

```

The third and last iterator is the REPEAT-UNTIL.

In this kind of iterator :
 we execute the instruction
 we test over the condition.

Let's have an example to make it more clear.

We suppose that we want the user to enter his age, but we have to make sure that the age is not negative or equal to 0

so we are going to ask him to enter his age, then we verify the age:

```

ALGORITHM iterative_repeat

VAR

age : INTEGER;

BEGIN

  REPEAT

    Read( age )

  UNTIL ( age >= 0 )

  Write( age )

END

```

Unlike the while-do, in this type of iterator, we make sure that the instructions will be executed at least once.

What is the value of x after executing this code.

```
count := 0; x := 3;
WHILE (count < 3) DO
    count := count + 1;
    y := (1+2*count) % 3;
    SWITCH (y) DO
        default :
            case 0 : x := x-1; BREAK;
            case 1 : x := x+5;
    END_SWITCH
END_WHILE
```

0

3

6

9

12

What is output of this code

```
i:=0;
j:=0;
WHILE (i < 10 AND j < 5) DO
    i := i+1;
    j := j+1;
END_WHILE
Write(i)
Write(j)
```

error

0

2

Conclusion

What you should know by now :

Recapitulation

Let's recap!

In this super skill, we have seen that algorithm is not only about programming. It's also about logical thinking and problem solving.

During this super skill, we have learned everything about algorithms and its elements:

How to create and design algorithms?

How an algorithm should be structured?

What are the variables and their types?

What are the three types of processing?

An algorithm is the way of thinking without taking in consideration everything linked to the machine problem like managing memory, correcting syntax.. etc

CHECKPOINT

Objective

At this checkpoint, you are asked to write an algorithm that read a sentence, which ends with a point, character by character, and to determine:

Instructions

- The length of the sentence (the number of characters).
- The number of words in the sentence (assuming that the words are separated by a single space).
- The number of vowels in the sentence.

You have to keep in mind that:

- Each character will be treated separately.
- The last character is the point.
- Use three variables as counters.