Algorithms

At the start, an algorithm is a series of steps to solve a problem. An algorithm should not be language specific, as the plan or steps should be able to be programmed in any language. That being said, there needs to be some standards which are used in the representation of algorithms, particularly in a teaching environment. While some of these standards may not match directly to how a structure might be coded in a particular language, these are what will be used in Information Technology examination papers (Paper 1 and Paper 2) and candidates are expected to follow these guidelines when answer both papers.

An algorithm can be represented in many ways, including flowcharts or pseudocode. The focus of this document is on pseudocode and its use to represent an algorithm.

1. Variables

In an algorithm, variables do not need to be defined or typed. This is because this process varies depending on language. However, variables do need to be named and assigned values. The standard symbol to indicate a variable being assigned a value, either at initialization or as a result of a calculation is: ←

Examples:

An integer variable:	i ← 10	
A string variable:	name ← "Bob"	//note " " for strings
A character variable:	old ← 'y'	//note ' ' for characters
A real variable:	mass ← 25.63	
An array literal:	a[ ] ← 5 ; 6 ; 4 ; 3	
An individual array element:	graph[3] ← 15	
A calculation:	age ← age + 10	

2. Blocks of code

All blocks of code are indented for readability as would be the case in a program, starting from the left margin. Any structure, such as a loop or decision, starts an indented section. Such "inner blocks" have a begin and end structure. Certain structures including the start of a loop or the start of decision stand as the begin, but these will be a line indicating the end of the structure. These will be explained in examples.

3. Conditions, Boolean, arithmetic and logical operators

The standard set of logical operators is: > ; < ; >= ; <= ; <>
The standard set of Boolean operators is: NOT ; AND ; OR. These are written in upper case.
The standard set of arithmetic operators is: + ; - ; * ; / ; MOD

Examples:

```
x < 10
y > 25
z >= 50
k <> 30
```

These would generally be used in a condition statement such as:

```
if x < 10
if k <> 30
if z < 15 AND y <> 10
while x <5 OR k > 11
```

Calculation examples:

```
b ← d * 10
z ← d MOD 2
b ← b + a
```

Decisions and loops

The main structures which need to be considered are:

```
if ….. else statements
for …. loops
while ….. loops
case / switch structure
```

Examples:

```
if k < 10
      line of code
      line of code
      line of code
end if

if d > 15
      line of code
      line of code
      else
           line of code
           line of code
      end else
end if

for k ← 1 to 10, inc by 1
      line of code
      line of code
      line of code
end for

for k ← 20 to 0, dec by 2
      line of code
      line of code
end for
```

```
while x < 10
        line of code
        line of code
        line of code
end while
```
(This structure is equivalent in Java and Delphi)

A do …. while loop is an exception to the rule of not having an end! The while statement acts as the end of the structure.

```
do
        line of code
        line of code
        line of code
while x < 10
```
(This structure is equivalent to a “repeat…until” in Delphi)

```
case age of
        18 : line of code
        20 : line of code
end case

case temperature of
        36 : begin              //this structure requires its own begin
                line of code
                line of code
               end
        40 : line of code
end case
```
(This structure is the equivalent of a switch/case in Java)

Output of values, returning values, method calls

The simplest structures must, once again, be used here to be language independent. Method names are always followed by a pair of parentheses, whether there are parameters or not.

Examples:

```
display x
display “This text will appear as output”
display “He is ” + age + “ years old”              // + is used as a concatenator

return z
return (a / b)
return name
return “Danger ahead!”
return “Name: ” + nameVariable
```

```
thisMethod( a: string)              //method header with a parameter
b ← thisMethod("happy")             //method call, assuming b is of type string
```

String manipulation

Functions such as finding the length of a string, extracting characters from a string pose some difficulties due to the large variations between languages.

```
LENGTH / LEN
LEFT
RIGHT
MID
```

```
b ← length(stringVariable)
name ← LEFT(stringVariable,2)       //taking the leftmost 2 characters
clear ← MID(stringVariable,4,2)     //starting at position 4, taking 2 characters
d ← characterAt(string,3)           //finds a character at position 3
```

Lastly, any other functions which are very different from language to language, may be written out as an English equivalent.

Example algorithm

```
size ← 4
temp ← 0
runningAvg ← 0
for k ← 0 to size - 1 inc by 1
      temp ← temp + ageArr[k]
      count ++
      runningAvg ← runningAvg + (temp / count)
      if (runningAvg > 60)
        display "Error"
      end if
end for
display (temp / count)
```