Introduction to Systematic Programming
Unit 2 - Algorithms and Problem-solving

2.1 The nature of algorithm steps

A simple algorithm might contain only **unitary steps** (or 'indivisible' steps), performed one after another in the order stated, such as:

> "input next data value",
> "add 1 to this number",
> "write out result",
> etc.

but this isn't so in general. To see why, let's look at an example algorithm. It isn't for a 'computer-type' problem, but it will serve to illustrate some important points. Suppose you wished to phone a message to a Mr. X. Remembering that algorithms are supposed to guarantee achieving an objective, and therefore allowing for the possibility that the telephone line is (repeatedly) engaged, a possible algorithm could be:

```
IF X's phone number is not already known THEN
     look it up
END IF

WHILE we haven't yet got through LOOP

     pick up the receiver
     dial X's number

     IF we get a ringing tone THEN
          wait until phone is answered
          state the message
          put down the receiver
     ELSE
          put down the receiver
          wait a few minutes
     END IF

END LOOP
```

We can see that some of the steps are of a simple unitary nature, for example:

```
pick up the receiver
dial X's number
```

but others are **compound steps**, in the sense that they *contain* other step(s) which are selected or repeated. In the case of selection, step(s) are (or are not) to be performed according to some stated **condition** or **test**, and in the case of repetition, step(s) are to be repeated according to some stated condition or test.

2.1.1 Expressing selection in an algorithm

An operation frequently required when expressing algorithms is the ability to indicate whether a particular series of actions are to be performed, or are not to be performed, according to whether some condition is satisfied or not. Most programming languages contain such **selection steps**, and in line with our aim (expressed in Unit 1) of making the transfer of an algorithm into an Ada program as simple an operation as possible, we will use the same notation (or **syntax**) when expressing a selection step in our algorithm as is used in the Ada programming language, namely:

```
IF some condition {is true} THEN
     {perform only these} step(s)
ELSE
     {perform only these alternative} step(s)
END IF
```

The precise meaning (or **semantics**) of this selection step (which is often referred to as an **if..then..else..** step) is as follows:

> Look at (or **evaluate**) the condition and if (and only if) that condition is satisfied (ie. is `True`), carry out the first set of step(s) - but don't carry out the second set of step(s). Alternatively, if the condition was not satisfied (ie. was `False`), carry out the second set of step(s) - but don't carry out the first set of step(s). In either case, when the appropriate step(s) have been carried out, continue by performing the next step which follows after the `END IF`.

(Note that the braces {...} are not part of the notation; they simply enclose explanation).

We can see an example of this **if..then..else..** step in the previous algorithm, where, depending upon whether or not we get a ringing tone, we wish to perform either the actions necessary to state the message, or the actions which involve waiting a few minutes (before trying again).

Sometimes we don't need any action at all when the condition turns out not to be satisfied (ie. to be `False`), and in this is case we can use the shortened notation:

```
IF some condition {is true} THEN
     {perform these} step(s)
END IF
```

The precise meaning of this shortened selection step (which is often referred to as an **if..then..** step) is as follows:

> Evaluate the condition and if (and only if) that condition is satisfied (ie. is `True`), carry out the specified set of step(s). Alternatively, if the condition was not satisfied (ie. was `False`) don't carry out the specified set of step(s). In either case, whether the appropriate step(s) have been carried out or not, continue by performing the next step which follows the `END IF`.

We can also see an example of this **if..then..** step in the previous algorithm, where, when we do not already know Mr. X's telephone number, we perform the actions necessary to look up the telephone number.

2.1.2 Expressing repetition in an algorithm

Another operation frequently required when expressing algorithms is the ability to indicate that a particular series of actions are to be performed repeatedly, over and over again. Most programming languages contain several means for expressing such **repetition steps**, so we will start by using one of the most common forms. Again we will use the same notation for expressing a repetition step in our algorithm as is used in the Ada programming language, namely:

```
WHILE some condition {is true} LOOP
     {keep on doing these} step(s)
END LOOP
```

In precise terms, this repetition step (often referred to as a **while..loop..** step) means:

> Evaluate the condition, when the result is `True`, obey the controlled step(s) and then re-evaluate the condition to see if the repetition is to proceed. When the re-evaluated condition result is still `True`, obey the controlled step(s) again and then re-evaluate the condition again to see if the repetition is to proceed further, and so on. When at any evaluation or re-evaluation the condition result is `False`, stop the repetition (ie. terminate the entire while loop immediately), and continue by performing the next step which follows after the `END LOOP`.

We can see an example of this **while..loop..** step in the previous algorithm, where we pick up the telephone, dial the number, and attempt to deliver the message, and repeatedly keep on doing these step so long as we have not succeeded in delivering the message.

It is important to note that these compound steps can be **nested** arbitrarily. For example in the algorithm above one of the steps controlled by the `WHILE..LOOP..` repetition is an `IF..THEN..` step. Similarly, within such an `IF..THEN..` we could in turn write any further unitary or compound steps that our algorithm might need, and so on, without any specific limitation.

2.1.3 Layout to be used

There are no special rules regarding layout, except that whatever layout is chosen should attempt to make the structure of the algorithm as clear as possible. However the course material will adopt the convention that the keywords `IF`, `ELSE` and `END IF` of a selection, and `WHILE` and `END LOOP` of a repetition are aligned underneath one another, and that actions within selections and repetitions are indented from this alignment. The same layout arrangement should be used in the corresponding Ada program, where an indentation of three or four spaces is recommended.

2.1.4 Algorithm structure - summary

We have seen three forms of construction which may be used in expressing algorithms:

sequencing	-	implied by the order in which steps are written
selection	-	using `IF..THEN..` to choose between alternative actions, or possibly between some action(s) and no action at all
repetition	-	expressed using `WHILE..LOOP..`

These are extremely important because *all* algorithms can be expressed using suitable combinations of them. Since these constructions are so important, it is naturally the case that they are provided directly in most programming languages. Thus in Ada `IF..THEN..ELSE..END IF` or `IF..THEN..END IF` are used for expressing selection, and `WHILE..LOOP..END LOOP` for expressing repetition. These ideas also form the basis for our methodology for successfully producing correct programs. This course adopts a methodology usually called **structured programming**, and primarily uses a variation of the techniques referred to as **step-wise refinement** and **top-down design** as a means for obtaining such structured programs.

2.2 Values and variables

Nearly all computation (and hence nearly all algorithm specification) involves calculation on values. For example in averaging problems we add up a sequence of values to obtain another value, the total; divide this by a further value, the count of the numbers totalled; and hence obtain yet another value, the average. To be able to refer to values conveniently, we need to give them meaningful names - eg. `Total`, `Count` and `Average` respectively. This technique is vital in both algorithm specification and program writing, and a special term is used to describe such a named value - we call it a **variable** (just as in ordinary school algebra).

Given a variable, we can, *at different moments* in the execution of an algorithm, *associate different values* with that variable; but at *any given instant*, there can only be *one particular value* associated with the variable. As an illustration, consider the variables `Number` and `RunningTotal` in the following algorithm fragment:

```
1)   set RunningTotal to zero
2)   input a data value (call it Number)
3)   WHILE Number is not equal to zero LOOP
4)        add the value of Number to RunningTotal
5)        input the next data value (call it Number)
6)   END LOOP
7)   output the value of RunningTotal
```

Step (2) involves inputting the first data value and storing this value in the variable `Number`. The `WHILE..LOOP..` repetition then involves repeatedly testing the value stored in the variable `Number` and (if the value of `Number` is not zero) performing two steps. The first step (4) involves adding the value stored in `Number` to another variable `RunningTotal` (which has initially been set to zero at step (1)) and the second step involves inputting another data value and storing this value in the variable `Number`, overwriting the previous value with this new value. At some point a zero data value will be input to `Number` at step (5), and the subsequent test at step (3) will stop the repetition. When the repetition terminates, the value associated with `RunningTotal` will be the sum of the input data values (excluding the zero terminator value), so that at step (7) the sum of the data values input can be output.

For example, if we were to present the data values: 4 2 0

to the algorithm, the values of the variables at various stages in the algorithm would be as follows:

	`RunningTotal`	`Number`
After completion of steps (1) and (2)	0	4
The test at step (3) would be `True`		
After completion of step (4)	4	4
After completion of step (5)	4	2
The test at step (3) would be `True`		
After completion of step (4)	6	2
After completion of step (5)	6	0
The test at step (3) would be `False`, so the repetition stops		
Step (7) would output the current value of `RunningTotal`, ie the value 6		

2.3 Assignment and the "becomes" symbol (:=)

Notice that there is an alternative way of expressing step (4) above; we could have put:

```
set RunningTotal to RunningTotal + Number
```

This has exactly the same meaning, but it shows the use of arithmetic and the methods by which values are associated with variables more explicitly - it shows that what we are saying, in detail, is:

take the values currently associated with `RunningTotal` and `Number`

add these value together, giving a new value

finally associate this new value with `RunningTotal`

(And the previous value of `RunningTotal` is "lost" (or "over-written") - eg. if the value of `Number` was 2 and the value of `RunningTotal` was 4 before, it is now 6 and all record of its previous value has been lost).

A more compact way of expressing this type of instruction is to use a symbol instead of writing '`set...to...`'. The notation used by Ada is to leave out the word '`set`' and replace '`to`' with the symbol '`:=`'. Thus in the present case we would put:

```
RunningTotal := RunningTotal + Number
```

The symbol `:=` is best read as *becomes*; eg. here we would say "`RunningTotal` becomes `RunningTotal` plus `Number`". This sort of algorithm step is known as an **assignment step**, which **assigns** a new value to the variable `RunningTotal`.

Another example is: `Count := Count + 1`

Here the value obtained by adding `1` to the value associated with `Count` is assigned to the variable `Count`. Or, loosely but more simply, "`Count` becomes `Count` plus `1`".

2.4 Expressing input and output steps in Ada notation

The basic concept in performing input is obtaining the next available value from the data and associating it with a variable; for example in step (5) of the algorithm above we have written:

```
          input the next data value (call it Number)
```

The Ada notation for this is would be: `Get(Item => Number)`

The basic concept for output is sending a value to some destination (such as a VDU, terminal or a disc file). In step (7) of the above algorithm we have written:

```
          output the value of RunningTotal
```

The Ada notation for this would be: `Put(Item => RunningTotal)`

However, for output we don't necessarily have to specify a variable - in fact anything that produces a value that can sensibly be printed will do; eg:

a) `Put(Item => First + Second)` - outputs the value of `First + Second`, assuming `First` and `Second` are variables

b) `Put(Item => Average - 2.5)` - outputs the value of `Average - 2.5`, assuming that `Average` is a variable

c) `Put(Item => 2 * Length + 1)` - outputs the value of `2 □ Length + 1`, assuming that `Length` is a variable, and that `*` stands for multiplication

d) `Put(Item => "Total value was:")`

(a) - (c) show the means of expressing numeric output, but (d) illustrates text or string output. Such a step states that the literal text string enclosed between the quote marks is to be written to the output device. This is very important in practice, for it will allow us (for example) to instruct the computer to produce messages (such as: "Type two numbers") and to output text which explains the meaning of numeric output (such as in (d) above and in what follows).

Note that we often need several successive output steps in an algorithm, eg:

```
      Put(Item => "The sum of ")
      Put(Item => First)
      Put(Item => " and ")
      Put(Item => Second)
      Put(Item => " is ")
      Put(Item => First + Second)
```

would be necessary to produce the output:

```
      The sum of 345 and 123 is 468
```

(Assuming, for example, that the variables `First` and `Second` held the values `345` and `123` respectively).

2.5 Example

As an example of the construction of a computer algorithm, consider the following problem.

The Problem:

Given data comprising a list of pairs of values representing the number of items sold and the corresponding unit price (or price per item) of a series of sales at a supermarket check-out, the list being of unknown length but terminated by two zero values, calculate the total amount of the all sales and the number of sales which exceeded £100.

The Solution:

i) **Think through the logic of the problem**

Need to input each pair of values in turn and for each pair calculate the sale value, add the sale value to a running total, and add 1 to a count (of the number high valued sales) if the sales value exceeds £100.

=> Need a repetition where each pair of values is input and processed on each action step of the repetition

ii) **Outline a scheme for solving the problem**

Following the example in [2.2] we see that this problem will need the same form of repetition. Thus:

```
set SalesTotal to zero
set HighSaleCount to zero
input two data values (call them NumberSold and UnitPrice)

WHILE NumberSold and UnitPrice are not both zero LOOP

     set SaleValue to NumberSold × UnitPrice
     add SaleValue to SalesTotal
     IF SaleValue > £100 THEN
          add 1 to HighSaleCount
     END IF
     input next two data values (call them NumberSold and UnitPrice)

END LOOP

output SalesTotal and HighSaleCount
```

Notice that this utilises an algorithm fragment of the form:

```
input first value
WHILE value does not signal termination LOOP
     use that value in some calculation
     input the next value
END LOOP
```

but where the '`value`' referred to in the fragment has been replaced by references to `NumberSold` and `UnitPrice` as the pair of data values being processed on each repetition. Notice too that the example algorithm in [2.2] also utilises this algorithm fragment, but where '`value`' was replaced by `Number`.

This algorithm fragment or **subalgorithm** is important because **many parts of computer algorithms have this structure**. Be sure you understand and remember it; and also appreciate when it's relevant to use it, ie. whenever inputting and processing a list of data values which is terminated by some special marker or sentinel value. Part of the technique of learning to program is that of remembering such subalgorithms, and being able to identify situations where they may be reused in an identical or very similar form. You will see many such subalgorithms during the lectures, tutorials and case studies of this course.

iii) **Complete the algorithm using Ada-like steps for input, output and assignment**

```
SalesTotal := 0
HighSaleCount := 0
Get(Item => NumberSold)
Get(Item => UnitPrice)

WHILE NumberSold and UnitPrice are not both zero LOOP

     SaleValue := NumberSold * UnitPrice
     SalesTotal := SalesTotal + SaleValue
     IF SaleValue > £100 THEN
          HighSaleCount := HighSaleCount + 1
     END IF
     Get(Item => NumberSold)
     Get(Item => UnitPrice)
END LOOP

Put(Item => "Total sales was ")
Put(Item => SalesTotal)
Put(Item => "Number of sales exceeding £100 was ")
Put(Item => HighSaleCount)
```

In the example program in [1.6] we observed that an essential component of an Ada program is to specify what variables are needed in the program, so that when writing an algorithm it is always helpful to note what variables have been used. In the above algorithm we have used the variables:

`NumberSold`	- to hold the number of items sold (in each sale in turn)
`UnitPrice`	- to hold the unit price of the item sold (in each sale in turn)
`SaleValue`	- to hold the value of a sale (for each sale in turn)
`SalesTotal`	- to hold the running total of all the sales values
`HighSaleCount`	- to keep a count of the number of sales exceeding £100

Notice that part of this algorithm has been expressed in normal text and part in italics. In future when designing algorithms we will use normal text to denote those parts which are already expressed in Ada-like notation, and italics for those parts which require further alteration before they can be used directly in an Ada program. From this example we see that effectively all but a part of two lines of the algorithm is already written in the Ada language - perhaps programming isn't so difficult after all!

2.6 General advice on how to set about writing an algorithm

Try to follow this pattern:

i) Quickly think through the whole problem, trying to get a general overall picture of what kinds of action are required to solve the problem. It is often helpful to think about what input data is available, about what output results are required and about their form (what order do data items come in/what order must results be output in?)

ii) Write down the main actions involved (using terse English descriptions); so far as possible in the correct order. In particular try to identify any repetitions that are involved in the overall structure of the problem.

iii) Now write out a (computer) algorithm, using our Ada-style notation. Remember to deal with input and output, which must be fully specified in computer algorithms, and must be done in a way that is compatible with the physical structure of input/output devices (ie. working left-to-right and downwards, so that, in particular, any given data item can be read in only once). For now, leave anything that is not expressible in our formal style in terse English or some other suitable (eg. mathematical) notation (whichever seems more appropriate). However, always try to use one of the formalisms `IF..THEN..END IF` or `IF..THEN..ELSE..END IF` for expressing any selections, and the formalism `WHILE..LOOP..END LOOP` for expressing any repetitions.

These are general hints, not precise rules, and at this stage you cannot expect to be more than moderately adept at applying them, but programming is a practical discipline, and with practical experience, you will steadily increase your skill.

Occasionally you may not be able to make any progress with step (i). If this happens, or if you get stuck at a later point, don't give up, but ask these questions:

- If *I* had to do the work that this problem requires by hand, how would *I* do it?

- What arithmetic is needed (what numbers would be used, what totals, what intermediate results, etc.)?

- Is there anything which I would have to repeat a number of times (and if so, how would I know when to stop)?

You should find that if you can see, and can state in reasonably precise detail, how one could solve a problem with pencil and paper, then this is a large step towards expressing the solution as a computer algorithm.