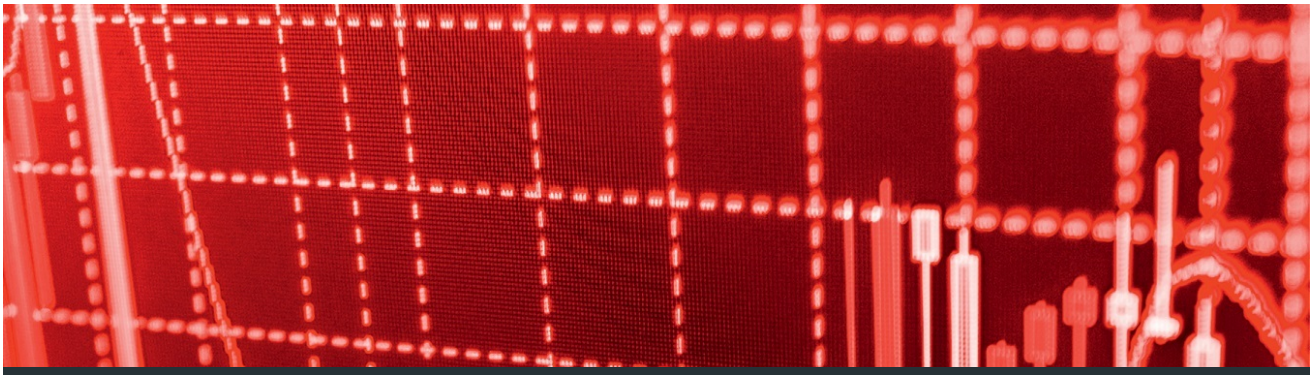
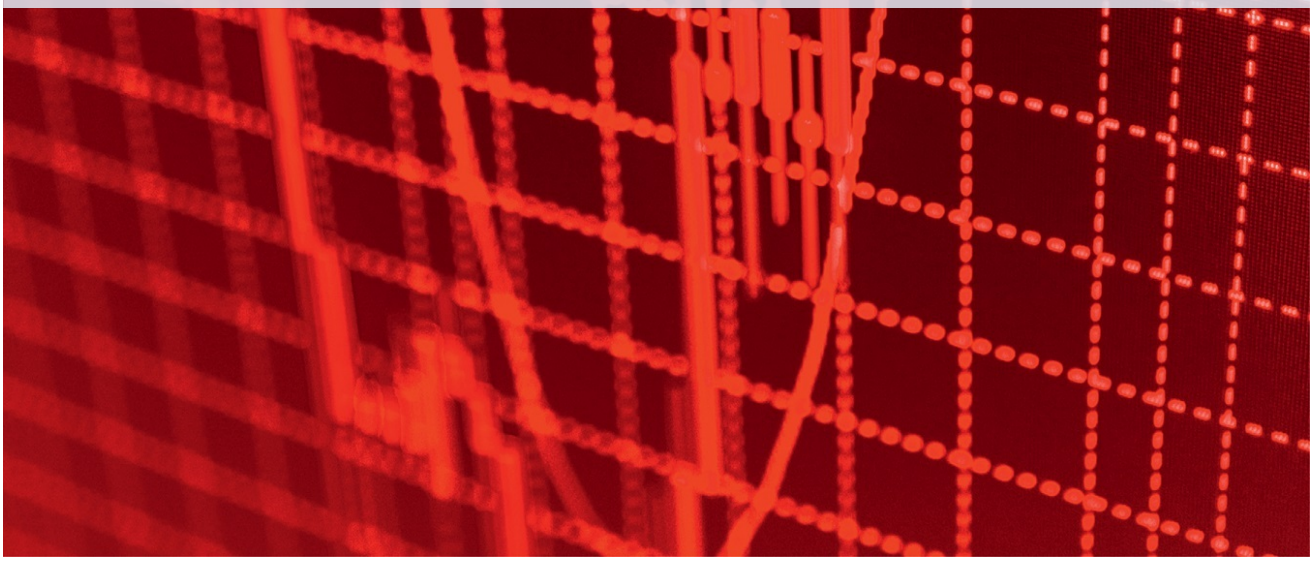


Chapter 12:
Algorithm design and problem-solving

Learning objectives

By the end of this chapter you should be able to:

- show an understanding of abstraction
- describe the purpose of abstraction
- produce an abstract model of a system by only including essential details
- describe and use decomposition
- break down problems into sub-problems leading to the concept of a program module
- show understanding that an algorithm is a solution to a problem expressed as a sequence of defined steps
- use suitable identifier names for the representation of data used by a problem and represent these using an identifier table
- write pseudocode that contains input, process and output
- write pseudocode using the four basic constructs of assignment, sequence, selection and repetition
- document a simple algorithm using pseudocode
- write pseudocode from a structured English description or a flowchart
- describe and use the process of stepwise refinement to express an algorithm to a level of detail from which the task may be programmed
- use logic statements to define parts of an algorithm solution.



12.01 What is computational thinking?

Computational thinking is a problem-solving process where a number of steps are taken in order to reach a solution. It is a logical approach to analysing a problem, producing a solution that can be understood by humans and used by computers.

Computational thinking involves five key strands: abstraction, decomposition, data modelling, pattern recognition and algorithmic thinking.

Abstraction

Abstraction involves filtering out information that is not necessary to solve a problem. There are many examples in everyday life where abstraction is used. Figure 12.02 shows part of the underground map of London, UK. The purpose of this map is to help people plan their journey within London. The map does not show a geographical representation of the tracks of the underground train network nor does it show the streets above ground. It shows the stations and which train lines connect the stations. In other words, the information that is not necessary when planning how to get from one landmark to another is filtered out. The essential information we need to be able to plan our route is clearly represented.

Abstraction gives us the power to deal with complexity. An algorithm is an abstraction of a process that takes inputs, executes a sequence of steps, and produces outputs. An abstract data type defines an abstract set of values and operations for manipulating those values.

TASK 12.01

Use the aerial photograph in Figure 12.01 and draw a map just showing the essential details for finding a route from landmark A to landmark B.

Figure 12.01 Aerial photograph of part of a city

Decomposition

Decomposition means breaking problems down into sub-problems in order to explain a process more clearly. Decomposition leads us to the concept of program modules and using procedures and functions.

Data modelling

Data modelling involves analysing and organising data (see Chapter 13). We can set up abstract data types to model real-world concepts, such as queues or stacks. When a programming language does not have such data types built-in, we can define our own by building them from existing data types. There are more ways to build data models. In Chapter 27 we cover object-oriented programming where we build data models by defining classes. In Chapter 29 we model data using facts and rules. In Chapter 26 we cover random files.

Pattern recognition

Pattern recognition means looking for patterns or common solutions to common problems and using these to complete tasks in a more efficient and effective way. There are many standard algorithms to solve standard problems, such as sorting (see Section 13.03 and 23.03) or searching (see Section 13.02 and 23.04).

Algorithm design

Algorithm design involves developing step-by-step instructions to solve a problem.

12.02 What is an algorithm?

We use algorithms in everyday life. If you need to change a wheel on a car, you might need to follow instructions (the algorithm) from a manual.

1 Take a spanner and loosen the wheel nuts.

2 Position a jack in an appropriate place.

3 Raise the car.

4 Take off the wheel nuts and the wheel.

5 Lift replacement wheel into position.

6 Replace wheel nuts and tighten by hand.

7 Lower the car.

8 Fully tighten wheel nuts.

This might sound all very straightforward. However, if the instructions are not followed in the correct logical sequence, the process might become much more difficult or even impossible. For example, if you tried to do Step 1 after Step 3, the wheel may spin and you can't loosen the wheel nuts. You can't do Step 4 before Step 3.

If you want to bake a cake, you follow a recipe.

1 Measure the following ingredients: 200g sugar, 200g butter, 4 eggs, 200g flour, 2 teaspoons baking powder and 2 tablespoons of milk.

2 Mix the ingredients together in a large bowl, until the consistency of the mixture is smooth.

3 Pour the mixture into a cake tin.

4 Bake in the oven at 190° C for 20 minutes.

5 Check it is fully cooked.

6 Turn cake out of the tin and cool on a wire rack.

The recipe is an algorithm. The ingredients are the input and the cake is the output. The process is mixing the ingredients and cooking the mixture in the oven.

Sometimes a step might need breaking down into smaller steps. For example, Step 2 can be more detailed.

2.1 Beat the sugar and butter together until fluffy.

2.2 Add the eggs, one at a time, mixing constantly.

2.3 Sieve the flour and baking powder and stir slowly into the egg mixture.

2.4 Add milk and mix to give a creamy consistency.

Sometimes there might be different steps depending on some other conditions. For example, consider how to get from one place to another using the map of the London Underground system in Figure 12.02.

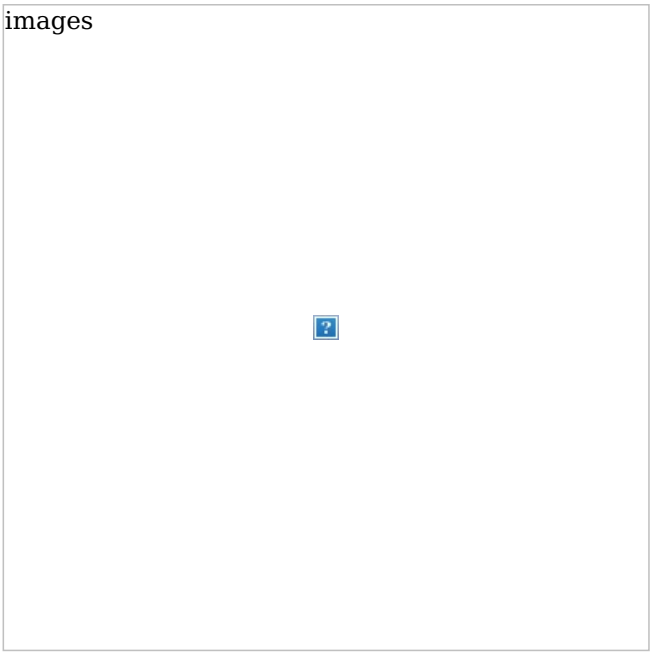


Figure 12.02 Underground map of London, UK

To travel from King's Cross St. Pancras to Westminster, we consider two routes:

- Route A: Take the Victoria Line to Green Park (4 stations); then take the Jubilee Line to Westminster (1 station)
- Route B: Take the Piccadilly Line to Green Park (6 stations); then take the Jubilee Line to Westminster (1 station).

Route A looks like the best route. If there are engineering works on the Victoria Line and trains are delayed, Route B might turn out to be the quicker route.

The directions on how to get from King's Cross St. Pancras to Westminster can be written as:

IF there are engineering works on the Victoria Line

THEN

Take the Piccadilly Line to Green Park (6 stations)

Take the Jubilee Line to Westminster (1 station)

ELSE

Take the Victoria Line to Green Park (4 stations)

Take the Jubilee Line to Westminster (1 station)

TASK 12.02

Write the steps to be followed to:

- make a sandwich
- walk from your school/college to the nearest shop
- log on to your computer.

Many problems have more than one solution. Sometimes it is a personal preference which solution to choose. Sometimes one solution will be measurably better than another.

12.03 Expressing algorithms

TIP

Computer scientists are interested in finding good solutions. A good solution gives the correct results, takes up as little computer memory as possible and executes as fast as possible. The solution should be concise, elegant and easy to understand.

In computer science, when we design a solution to a problem we express the solution (the algorithm) using sequences of steps written in **structured English** or **pseudocode**. Structured English is a subset of the English language and consists of command statements. Pseudocode resembles a programming language without following the syntax of a particular programming language. A **flowchart** is an alternative method of representing an algorithm. A flowchart consists of specific shapes, linked together.

An algorithm consists of a sequence of steps. Under certain conditions we may wish not to perform some steps. We may wish to repeat a number of steps. In computer science, when writing algorithms, we use four basic types of construct.

- **Assignment:** a value is given a name (identifier) or the value associated with a given identifier is changed.
- **Sequence:** a number of steps are performed, one after the other.
- **Selection:** under certain conditions some steps are performed, otherwise different (or no) steps are performed.
- **Repetition:** a sequence of steps is performed a number of times. This is also known as iteration or looping.

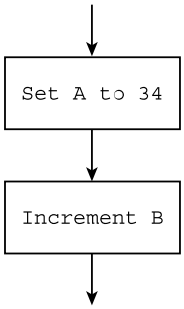
Many problems we try to solve with a computer involve data. The solution involves inputting data to the computer, processing the data and outputting results (as shown in Figure 12.03).

Figure 12.03 Input–process–output

We therefore also need input and output statements.

We need to know the constructs so we know how detailed our design has to be. These constructs are represented in each of the three notations as shown in Table 12.01.

In this book, algorithms and program code are typed using the `Courier` font.

	Structured English	Pseudocode	Flowchart
Assignment and Sequence	`SET A TO 34` `INCREMENT B`	`A ← 34` `B ← B + 1`	

Selection	`IF A IS GREATER THAN B`
	`  THEN ...`
	`  ELSE ...`
		`IF A > B`
		`  THEN ...`
		`  ELSE ...`
		`ENDIF`	

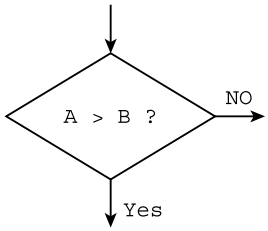
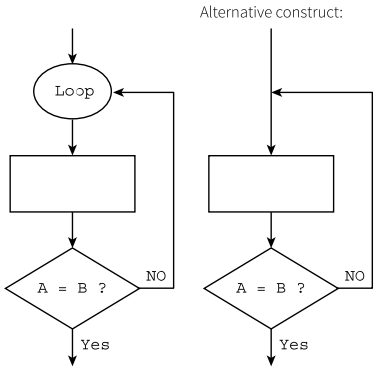
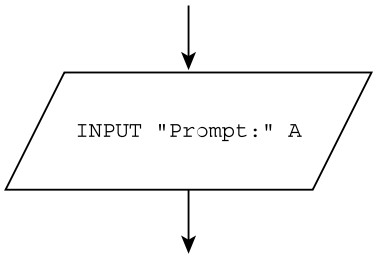
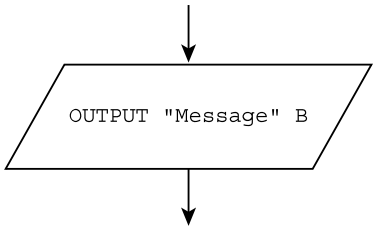
			
Repetition	REPEAT UNTIL A IS EQUAL TO B ...	REPEAT ... UNTIL A = B	
Input	INPUT A	INPUT "Prompt: " A	
Output	OUTPUT "Message" OUTPUT B	OUTPUT "Message", B	

Table 12.01 Constructs for computing algorithms

12.04 Variables

When we input data for a process, individual values need to be stored in memory. We need to be able to refer to a specific memory location so that we can write statements of what to do with the value stored there. We refer to these named memory locations as **variables**. You can imagine these variables like boxes with name labels on them. When a value is input, it is stored in the box with the specified name (identifier) on it.

For example, the variable used to store a count of how many guesses have been made in a number guessing game might be given the identifier `NumberOfGuesses` and the player's name might be stored in a variable called `ThisPlayer`, as shown in Figure 12.04.

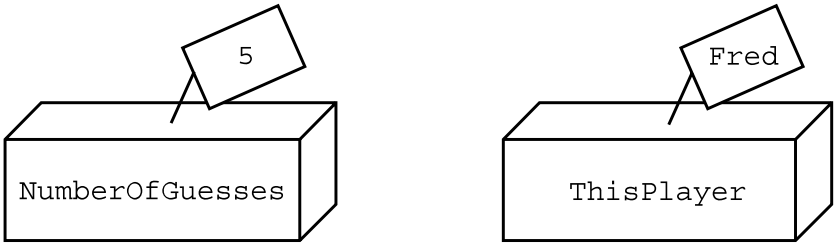


Figure 12.04 Variables

Variable identifiers should not contain spaces, only letters, digits and _ (the underscore symbol). To make algorithms easier to understand, the naming of a variable should reflect the variable's use. This means often that more than one word is used as an identifier. The formatting convention used here is known as CamelCaps. It makes an identifier easier to read.

12.05 Assignments

Assigning a value

The following pseudocode stores the value that is input (for example 15) in a variable with the identifier `Number` (see Figure 12.05(a)).

```
INPUT Number
```

The following pseudocode stores the value 1 in the variable with the identifier `NumberOfGuesses` (see Figure 12.05(b)).

```
NumberOfGuesses ← 1
```

Figure 12.05 Variables being assigned a value

Updating a value

The following pseudocode takes the value stored in `NumberOfGuesses` (see Figure 12.06 (a)), adds 1 to that value and then stores the new value back into the variable `NumberOfGuesses` (see Figure 12.06 (b)).

```
NumberOfGuesses ← NumberOfGuesses + 1
```

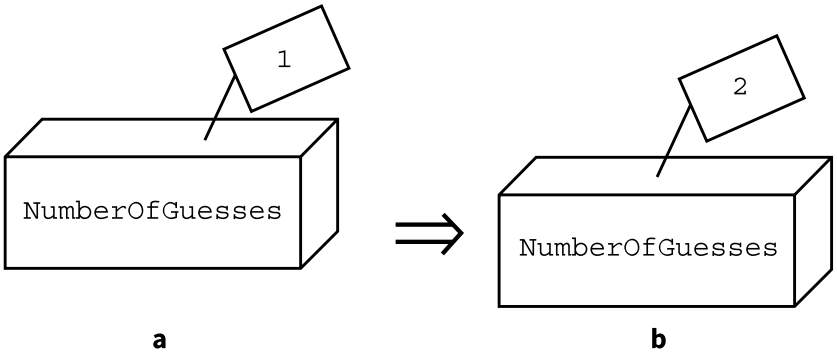


Figure 12.06 Updating the value of a variable

Copying a value

Values can be copied from one variable to another.

The following pseudocode takes the value stored in `Value1` and copies it to `Value2` (see Figure 12.07).

```
    Value2 ← Value1
```

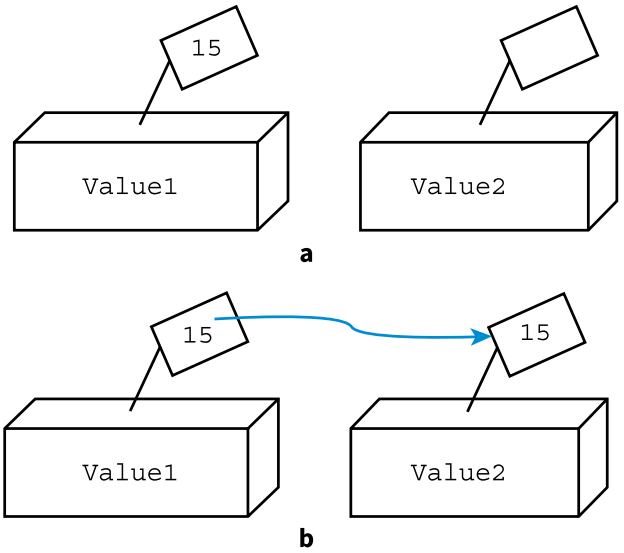


Figure 12.07 Copying the value of a variable

The value in `Value1` remains the same until it is assigned a different value.

Swapping two values

If we want to swap the contents of two variables, we need to store one of the values in another variable temporarily. Otherwise the second value to be moved will be overwritten by the first value to be moved.

In Figure 12.08(a), we copy the content from `Value1` into a temporary variable called `Temp`. Then we copy the content from `Value2` into `Value1` Figure 12.08(b)). Finally, we can copy the value from `Temp` into `Value2` (Figure 12.08(c)).

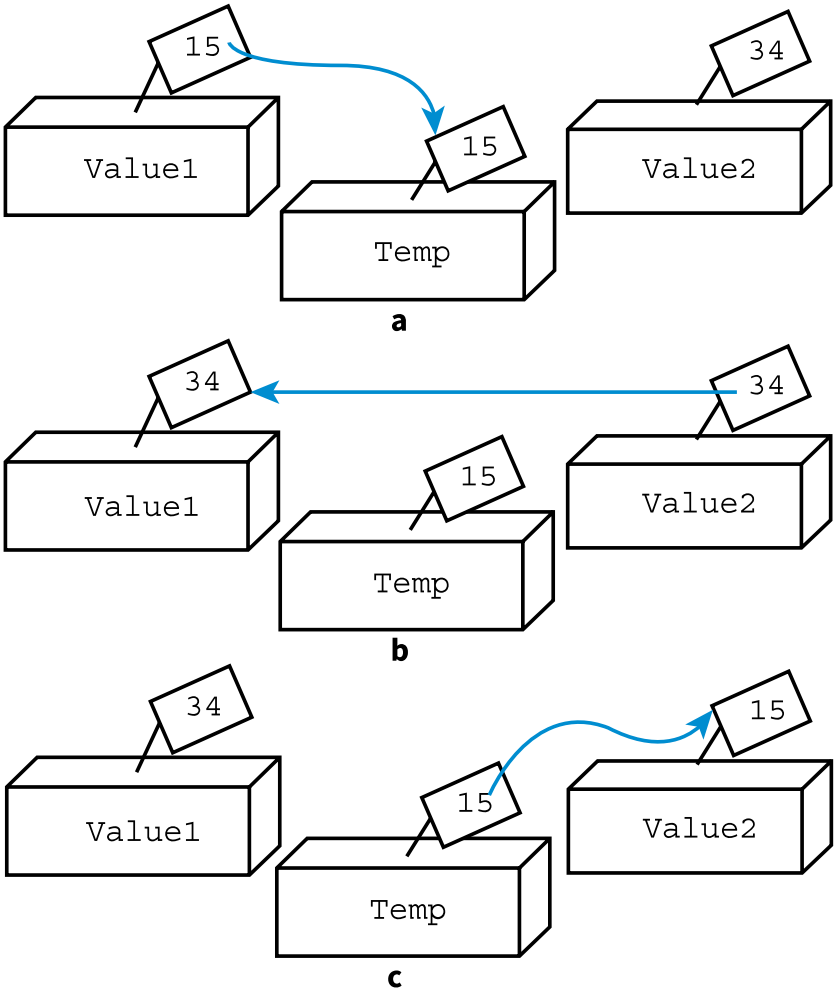


Figure 12.08 Swapping the values of two variables

Using pseudocode we write:

```
Temp ← Value1
Value1 ← Value2
Value2 ← Temp
```

WORKED EXAMPLE 12.01

Using input, output, assignment and sequence constructs

The problem to be solved: Convert a distance in miles and output the equivalent distance in km.

Step 1: Write the problem as a series of structured English statements:

```
INPUT number of miles
Calculate number of km
OUTPUT calculated result as km
```

Step 2: Analyse the data values that are needed.

We need a variable to store the original distance in miles and a variable to store the result of multiplying the number of miles by 1.61. It is helpful to construct an **identifier table** to list the variables.

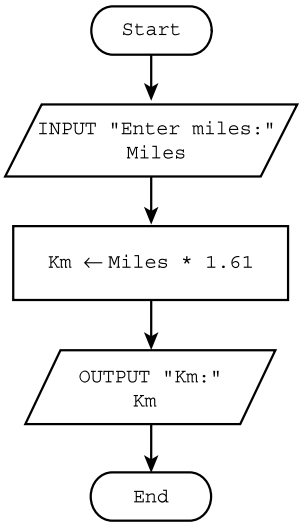
Identifier	Explanation
Miles	Distance as a whole number of miles
Km	The result from using the given formula: Km = Miles * 1.61

Table 12.02 Identifier table for miles to km conversion

Step 3: Provide more detail by drawing a flowchart or writing pseudocode.

The detail given in a flowchart should be the same as the detail given in pseudocode. It should use the basic constructs listed in Table 12.01.

Figure 12.09 represents our algorithm using a flowchart and the equivalent pseudocode.




```
INPUT "Enter miles:" Miles
Km ← Miles * 1.61
OUTPUT "km:", Km
```

Figure 12.09 Flowchart and pseudocode for miles to km conversion

TASK 12.03

Consider the following algorithm steps.

1 Input a length in inches.

2 Calculate the equivalent in centimetres.

3 Output the result.

List the variables required in an identifier table.

Write pseudocode for the algorithm.

12.06 Logic statements

In Section 12.02, we looked at an algorithm with different steps depending on some other condition:

IF there are engineering works on the Victoria Line

THEN

Take the Piccadilly Line to Green Park (6 stations)

Take the Jubilee Line to Westminster (1 station)

ELSE

Take the Victoria Line to Green Park (4 stations)

Take the Jubilee Line to Westminster (1 station)

The selection construct in Table 12.01 uses a condition to follow either the first group of steps or the second group of steps (see Figure 12.10).

A condition consists of at least one logic proposition (see Chapter 4, Section 4.01). Logic propositions use the relational (comparison) operators shown in Table 12.03.

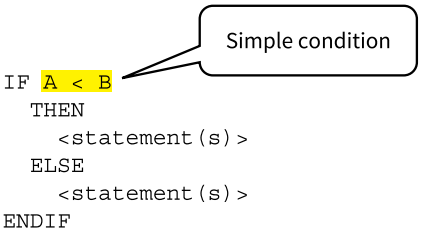


Figure 12.10 Pseudocode for the selection construct

Operator	Comparison
=	Is equal to
<	Is less than
>	Is greater than
<=	Is less than or equal to
>=	Is greater than or equal to
<>	Is not equal to

Table 12.03 Relational operators

Conditions are either TRUE or FALSE. In pseudocode, we distinguish between the relational operator = (which tests for equality) and the assignment symbol ←.

A person is classed as a child if they are under 13 and as an adult if they are over 19. If they are between 13 and 19 inclusive they are classed as teenagers. We can write these statements as logic statements.

- If Age < 13 then person is a child.
- If Age > 19 then person is an adult.
- If Age >= 13 AND Age <= 19 then person is a teenager.

TASK 12.04

A town has a bus service where passengers under the age of 12 and over the age of 60 do not need to pay a fare. Write the logic statements for free fares.

A number-guessing game follows different steps depending on certain conditions. Here is a description of the algorithm.

- The player inputs a number to guess the secret number stored.
- If the guess was correct, output a congratulations message.
- If the number input was larger than the secret number, output message “secret number is smaller”.
- If the number input was smaller than the secret number, output message “secret number is greater”.

We can re-write the number-guessing game steps as an algorithm in pseudocode:

```
SET value for secret number
INPUT Guess
IF Guess = SecretNumber
   THEN
      OUTPUT "Well done. You have guessed the secret number"
   ELSE
      IF Guess > SecretNumber
        THEN
          OUTPUT "secret number is smaller"
        ELSE
          OUTPUT "secret number is greater"
      ENDIF
ENDIF
```

More complex conditions can be formed by using the logical operators AND, OR and NOT. For example, the number-guessing game might allow the player multiple guesses; if the player has not guessed the secret number after 10 guesses, a different message is output.

```
IF Guess = SecretNumber
  THEN
    OUTPUT "Well done. You have guessed the secret number"
  ELSE
    IF Guess <> SecretNumber AND NumberofGuesses = 10
      THEN
        OUTPUT "You still have not guessed the secret number"
      ELSE
        IF Guess > SecretNumber
          THEN
            OUTPUT "secret number is smaller"
          ELSE
            OUTPUT "secret number is greater"
        ENDIF
    ENDIF
ENDIF
```

WORKED EXAMPLE 12.02

Using selection constructs

The problem to be solved: Take three numbers as input and output the largest number.

There are several different methods (algorithms) to solve this problem. Here is one method.

1. Input all three numbers at the beginning.
2. Store each of the input values in a separate variable (the identifiers are shown in Table 12.04).
3. Compare the first number with the second number and then compare the bigger one of these with the third number.
4. The bigger number of this second comparison is output.

See Worked Example 12.03 for another solution.

Identifier	Explanation

Number1	The first number to be input
Number2	The second number to be input
Number3	The third number to be input

Table 12.04 Identifier table for biggest number problem

The algorithm can be expressed in the following pseudocode:

```
INPUT Number1
INPUT Number2
INPUT Number3
IF Number1 > Number2
  THEN
  // Number1 is bigger
    IF Number1 > Number3
      THEN
        OUTPUT Number1
      ELSE
        OUTPUT Number3
    ENDIF
  ELSE
  // Number2 is bigger
    IF Number2 > Number3
      THEN
        OUTPUT Number2
      ELSE
        OUTPUT Number3
    ENDIF
ENDIF
```

When an `IF` statement contains another `IF` statement, we refer to these as **nested `IF` statements**.

Question 12.01

What changes do you need to make to output the smallest number?

WORKED EXAMPLE 12.03

Using selection constructs (alternative method)

The problem to be solved: Take three numbers as input and output the largest number.

This is an alternative method to Worked Example 12.02.

1. Input the first number and store it in `BiggestSoFar`
2. Input the second number and compare it with the value in `BiggestSoFar`.
3. If the second number is bigger, assign its value to `BiggestSoFar`
4. Input the third number and compare it with the value in `BiggestSoFar`
5. If the third number is bigger, assign its value to `BiggestSoFar`
6. The value stored in `BiggestSoFar` is output.

The identifiers required for this solution are shown in Table 12.05.

Identifier	Explanation
BiggestSoFar	Stores the biggest number input so far

`NextNumber`	The next number to be input

Table 12.05 Identifier table for the alternative solution to the biggest number problem

The algorithm can be expressed in the following pseudocode:

```
INPUT BiggestSoFar
INPUT NextNumber
IF NextNumber > BiggestSoFar
   THEN
      BiggestSoFar ← NextNumber
ENDIF
INPUT NextNumber
IF NextNumber > BiggestSoFar
   THEN
      BiggestSoFar ← NextNumber
ENDIF
OUTPUT BiggestSoFar
```

Note that when we input the third number in this method the second number gets overwritten as it is no longer needed.

There are several advantages of using the method in Worked Example 12.03 compared to the method in Worked Example 12.02.

- Only two variables are used.
- The conditional statements are not nested and do not have an ELSE part. This makes them easier to understand.
- This algorithm can be adapted more easily if further numbers are to be compared (see Worked Example 12.04).

The disadvantage of the method in Worked Example 12.03 compared to the method in Worked Example 12.02 is that there is more work involved with this algorithm. If the second number is bigger than the first number, the value of `BiggestSoFar` has to be changed. If the third number is bigger than the value in `BiggestSoFar` then the value of `BiggestSoFar` has to be changed again. Depending on the input values, this could result in two extra assignment instructions being carried out.

12.07 Loops

Look at the pseudocode algorithm in Worked Example 12.03. The two `IF` statements are identical. To compare 10 numbers, we would need to write this statement nine times. Moreover, if the problem changed to having to compare, for example, 100 numbers, our algorithm would become very tedious. If we use a repetition construct (a loop) we can avoid writing the same lines of pseudocode over and over again.

WORKED EXAMPLE 12.04

Repetition using REPEAT...UNTIL

The problem to be solved: Take 10 numbers as input and output the largest number.

We need one further variable to store a counter, so that we know when we have compared 10 numbers.

Identifier	Explanation
BiggestSoFar	Stores the biggest number input so far
NextNumber	The next number to be input
`Counter`	Stores how many numbers have been input so far

Table 12.06 Identifier table for the biggest number problem using `REPEAT...UNTIL`

The algorithm can be expressed in the following pseudocode:

```
INPUT BiggestSoFar
Counter ← 1
REPEAT
    INPUT NextNumber
    Counter ← Counter + 1
    IF NextNumber > BiggestSoFar
      THEN
        BiggestSoFar ← NextNumber
    ENDIF
UNTIL Counter = 10
OUTPUT BiggestSoFar
```

Note that when we input the next number in this method the previous number gets overwritten as it is no longer needed.

Question 12.02

What changes do you need to make to the algorithm in Worked Example 12.04:

a to compare 100 numbers?

b to take as a first input the number of numbers to be compared?

There is another loop construct that does the counting for us: the `FOR...NEXT` loop.

WORKED EXAMPLE 12.05

Repetition using FOR...NEXT

The problem to be solved: Take 10 numbers as input and output the largest number.

We can use the same identifiers as in Worked Example 12.04. Note that the purpose of `Counter` has changed.

Identifier	Explanation
BiggestSoFar	Stores the biggest number input so far
NextNumber	The next number to be input
Counter	Counts the number of times round the loop

Table 12.07 Identifier table for biggest number problem using a FOR loop

The algorithm can be expressed in the following pseudocode:

```
INPUT BiggestSoFar
FOR Counter ← 2 TO 10
      INPUT NextNumber
      IF NextNumber > BiggestSoFar
        THEN
          BiggestSoFar ← NextNumber
      ENDIF
NEXT Counter
OUTPUT BiggestSoFar
```

The first time round the loop, Counter is set to 2. The next time round the loop, Counter has automatically increased to 3, and so on. The last time round the loop, Counter has the value 10.

A **rogue value** is a value used to terminate a sequence of values. The rogue value is of the same data type but outside the range of normal expected values.

WORKED EXAMPLE 12.06

Repetition using a rogue value

The problem to be solved: A sequence of non-zero numbers is terminated by 0. Take this sequence as input and output the largest number.

Note: In this example the rogue value chosen is 0. It is very important to choose a rogue value that is of the same data type but outside the range of normal expected values. For example, if the input might normally include 0 then a negative value, such as −1, might be chosen.

Look at Worked Example 12.05. Instead of counting the numbers input, we need to check whether the number input is 0 to terminate the loop. The identifiers are shown in Table 12.08.

Identifier	Explanation
BiggestSoFar	Stores the biggest number input so far
NextNumber	The next number to be input

Table 12.08 Identifier table for biggest number problem using a rogue value

A possible pseudocode algorithm is:

```
INPUT BiggestSoFar
REPEAT
    INPUT NextNumber
    IF NextNumber > BiggestSoFar
      THEN
        BiggestSoFar ← NextNumber
    ENDIF
UNTIL NextNumber = 0
OUTPUT BiggestSoFar
```

This algorithm works even if the sequence consists of only one non-zero input. However, it will not

work if the only input is 0. In that case, we don't want to perform the statements within the loop at all. We can use an alternative construct, the `WHILE...ENDWHILE` loop.

```
INPUT NextNumber
BiggestSoFar ← NextNumber
WHILE NextNumber <> 0 DO // sequence terminator not encountered
    INPUT NextNumber
    IF NextNumber > BiggestSoFar
      THEN
        BiggestSoFar ← NextNumber
    ENDIF
ENDWHILE
OUTPUT BiggestSoFar
```

Before we enter the loop, we check whether we have a non-zero number. To make this work for the first number, we store it in `NextNumber` and also in `BiggestSoFar`. If this first number is zero, we don't follow the instructions within the loop. For a non-zero first number this algorithm has the same effect as the algorithm using `REPEAT...UNTIL`.

WORKED EXAMPLE 12.07

Implementing the number-guessing game with a loop

Consider the number-guessing game again, this time allowing repeated guesses.

1. The player repeatedly inputs a number to guess the secret number stored.
2. If the guess is correct, the number of guesses made is output and the game stops.
3. If the number input is larger than the secret number, the player is given the message to input a smaller number.
4. If the number input is smaller than the secret number, the player is given the message to input a larger number.

The algorithm is expressed in structured English, as a flowchart and in pseudocode.

Algorithm for the number-guessing game in structured English:

```
SET value for secret number
REPEAT the following UNTIL correct guess
   INPUT guess
   count number of guesses
   COMPARE guess with secret number
   OUTPUT comment
OUTPUT number of guesses
```

We need variables to store the following values:

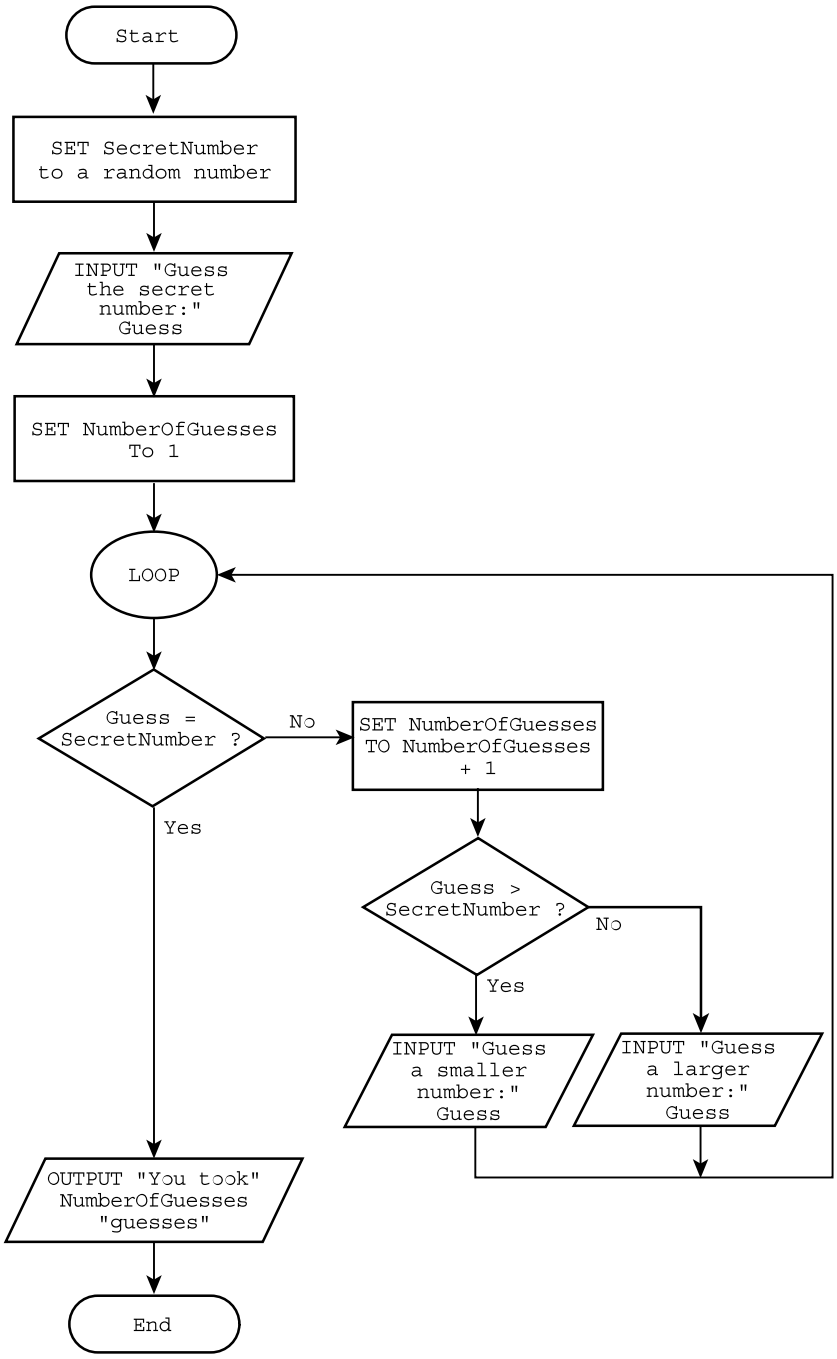
- the secret number (to be set as a random number)
- the number input by the player as a guess
- the count of how many guesses the player has made so far.

We represent this information in the identifier table shown in Table 12.09.

Identifier	Explanation
`SecretNumber`	The number to be guessed
`NumberOfGuesses`	The number of guesses the player has made
`Guess`	The number the player has input as a guess

Table 12.09 Identifier table for number-guessing game

Algorithm for the number-guessing game as a flowchart



Pseudocode for the number-guessing game with a post-condition loop

```
SecretNumber ← Random
NumberOfGuesses ← 0
REPEAT
    INPUT Guess
    NumberOfGuesses ← NumberOfGuesses + 1
    IF Guess > SecretNumber
      THEN
        // the player is given the message to input a smaller number
    ENDIF
    IF Guess < SecretNumber
      THEN
        // the player is given the message to input a larger number
```

```
    ENDIF
UNTIL Guess = SecretNumber
OUTPUT NumberOfGuesses
```

Pseudocode for the number-guessing game with a pre-condition loop

```
SecretNumber ← Random
INPUT Guess
NumberOfGuesses ← 1
WHILE Guess <> SecretNumber DO
    IF Guess > SecretNumber
      THEN
        // the player is given the message to input a smaller number
    ENDIF
    IF Guess < SecretNumber
      THEN
        // the player is given the message to input a larger number
    ENDIF
    INPUT Guess
    NumberOfGuesses ← NumberOfGuesses + 1
ENDWHILE
OUTPUT NumberOfGuesses
```

WORKED EXAMPLE 12.08

Calculating running totals and averages

The problem to be solved: Take 10 numbers as input and output the sum of these numbers and the average.

Identifier	Explanation
`RunningTotal`	Stores the sum of the numbers input so far
`Counter`	How many numbers have been input
`NextNumber`	The next number input
`Average`	The average of the numbers input

Table 12.10 Identifier table for running total and average algorithm

The following pseudocode gives a possible algorithm:

```
RunningTotal ← 0
FOR Counter ← 1 TO 10
      INPUT NextNumber
      RunningTotal ← RunningTotal + NextNumber
NEXT Counter
OUTPUT RunningTotal
Average ← RunningTotal / 10
OUTPUT Average
```

It is very important that the value stored in `RunningTotal` is initialised to zero before we start adding the numbers being input.

TIP

Which type of loop? If it is known how many repetitions are required, choose a FOR loop. If the statements inside the loop might never be executed, choose a WHILE loop. If the

statements inside the loop are to be executed at least once, a REPEAT loop might be more sensible.

TASK 12.05

Change the algorithm in Worked Example 12.08 so that the sequence of numbers is terminated by a rogue value of 0.

WORKED EXAMPLE 12.09

Using nested loops

The problem to be solved: Take as input two numbers and a symbol. Output a grid made up entirely of the chosen symbol, with the number of rows matching the first number input and the number of columns matching the second number input.

For example the three input values 3, 7 and &, result in the output:

&&&&&&&
&&&&&&&
&&&&&&&

We need two variables to store the number of rows and the number of columns. We also need a variable to store the symbol. We need a counter for the rows and a counter for the columns.

Identifier	Explanation
`NumberOfRows`	Stores the number of rows of the grid
`NumberOfColumns`	Stores the number of columns of the grid
`Symbol`	Stores the chosen character symbol
`RowCounter`	Counts the number of rows
`ColumnCounter`	Counts the number of columns

Table 12.11 Identifier table for the nested loop example

```
INPUT NumberOfRows
INPUT NumberOfColumns
INPUT Symbol
FOR RowCounter ← 1 TO NumberOfRows
    FOR ColumnCounter ← 1 TO NumberOfColumns
        OUTPUT Symbol // without moving to next line
    NEXT ColumnCounter
    OUTPUT Newline   // move to the next line
NEXT RowCounter
```

Each time round the outer loop (counting the number of rows) we complete the inner loop, outputting a symbol for each count of the number of columns. This type of construct is called a **nested loop**.

12.08 Stepwise refinement

Many problems that we want to solve are bigger than the ones we met so far. To make it easier to solve a bigger problem, we break the problem down into smaller steps. These might need breaking down further until the steps are small enough to solve easily.

For a solution to a problem to be programmable, we need to break down the steps of the solution into the basic constructs of sequence, assignment, selection, repetition, input and output.

We can use a method called **stepwise refinement** to break down the steps of our outline solution into smaller steps until it is detailed enough. In Section 12.02 we looked at a recipe for a cake. The step of mixing together all the ingredients was broken down into more detailed steps.

WORKED EXAMPLE 12.10

Drawing a pyramid using stepwise refinement

The problem to be solved: Take as input a chosen symbol and an odd number. Output a pyramid shape made up entirely of the chosen symbol, with the number of symbols in the final row matching the number input.

For example the two input values A and 9 result in the following output:

```
    A
   AAA
  AAAAA
 AAAAAAA
AAAAAAAAA
```

This problem is similar to Worked Example 12.09, but the number of symbols in each row starts with one and increases by two with each row. Each row starts with a decreasing number of spaces, to create the slope effect.

Our first attempt at solving this problem using structured English is:

```
01  Set up initial values
02  REPEAT
03      Output number of spaces
04      Output number of symbols
05      Adjust number of spaces and number of symbols to be output in next row
06  UNTIL the required number of symbols have been output in one row
```

The steps are numbered to make it easier to refer to them later.

This is not enough detail to write a program in a high-level programming language. Exactly what values do we need to set?

We need as input:

- the symbol character from which the pyramid is to be formed
- the number of symbols in the final row (for the pyramid to look symmetrical, this needs to be an odd number).

We need to calculate how many spaces we need in the first row. So that the slope of the pyramid is symmetrical, this number should be half of the final row's symbols. We need to set the number of symbols to be output in the first row to 1. We therefore need the identifiers listed in Table 12.12.

Identifier	Explanation
`Symbol`	The character symbol to form the pyramid
`MaxNumberOfSymbols`	The number of symbols in the final row

NumberOfSpaces	The number of spaces to be output in the current row
NumberOfSymbols	The number of symbols to be output in the current row

Table 12.12 Identifier table for pyramid example

Using pseudocode, we now refine the steps of our first attempt. To show which step we are refining, a numbering system is used as shown.

Step 01 can be broken down as follows:

```
01   // Set up initial values expands into:
01.1 INPUT Symbol
01.2 INPUT MaxNumberOfSymbols
01.3 NumberOfSpaces ← (MaxNumberOfSymbols – 1) / 2
01.4 NumberOfSymbols ← 1
```

Remember we need an odd number for `MaxNumberOfSymbols`. We need to make sure the input is an odd number. So we further refine Step 01.2:

```
01.2   // INPUT MaxNumberOfSymbols expands into:
01.2.1 REPEAT
01.2.2    INPUT MaxNumberOfSymbols
01.2.3 UNTIL MaxNumberOfSymbols MOD 2 = 1
01.2.4 // MOD 2 gives the remainder after integer division by 2
```

We can now look to refine Steps 03 and 04:

```
03 // Output number of spaces expands into:
03.1  FOR i ← 1 TO NumberOfSpaces
03.2    OUTPUT Space // without moving to next line
03.3  NEXT i
04    // Output number of symbols expands into:
04.1  FOR i ← 1 TO NumberOfSymbols
04.2    OUTPUT Symbol // without moving to next line
04.3  NEXT i
04.4  OUTPUT Newline // move to the next line
```

In Step 05 we need to decrease the number of spaces by 1 and increase the number of symbols by 2:

```
05   // Adjust values for next row expands into:
05.1 NumberOfSpaces ← NumberOfSpaces – 1
05.2 NumberOfSymbols ← NumberOfSymbols + 2
```

Step 06 essentially checks whether the number of symbols for the next row is now greater than the value input at the beginning.

```
06     UNTIL NumberOfSymbols > MaxNumberOfSymbols
```

We can put together all the steps and end up with a solution.

```
01     // Set Values
01.1   INPUT Symbol
01.2   // Input max number of symbols (an odd number)
01.2.1 REPEAT
01.2.2     INPUT MaxNumberOfSymbols
01.2.3 UNTIL MaxNumberOfSymbols MOD 2 = 1
01.3   NumberOfSpaces ← (MaxNumberOfSymbols – 1) / 2
01.4   NumberOfSymbols ← 1
02     REPEAT
03         // Output number of spaces
03.1       FOR i ← 1 TO NumberOfSpaces
```

```
03.2           OUTPUT Space // without moving to next line
03.3       NEXT i
04         // Output number of symbols
04.1       FOR i ← 1 TO NumberOfSymbols
04.2           OUTPUT Symbol // without moving to next line
04.3       NEXT i
04.4       OUTPUT Newline // move to the next line
05         // Adjust Values For Next Row
05.1       NumberOfSpaces ← NumberOfSpaces – 1
05.2       NumberOfSymbols ← NumberOfSymbols + 2
06     UNTIL NumberOfSymbols > MaxNumberOfSymbols
```

TASK 12.06

Use stepwise refinement to output a hollow triangle. For example the two input values A and 9 result in the following output:

```
    A
   A A
  A   A
 A     A
AAAAAAAAA
```

A first attempt at solving this problem using structured English is:

```
01   Set up initial values
02   REPEAT
03       Output leading number of spaces
04       Output symbol, middle spaces, symbol
05       Adjust number of spaces and number of symbols to be output in next row
06   UNTIL the required number of symbols have been output in one row
```

12.09 Modules

Another method of developing a solution is to decompose the problem into sub-tasks. Each sub-task can be considered as a 'module' that is refined separately. Modules are procedures and functions.

A **procedure** groups together a number of steps and gives them a name (an identifier). We can use this identifier when we want to refer to this group of steps. When we want to perform the steps in a procedure we call the procedure by its name.

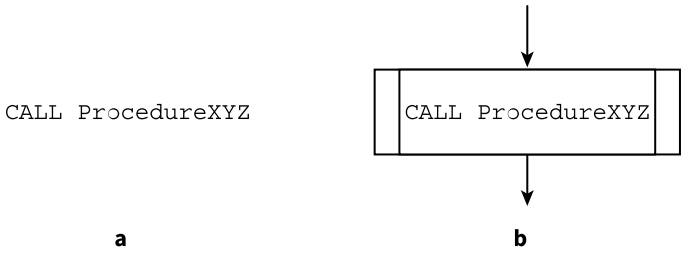


Figure 12.11 Representation of a procedure in (a) pseudocode and (b) a flowchart

A **function** groups together a number of steps and gives them a name (an identifier). These steps produce and return a value that is used in an expression. Worked Example 12.12 uses functions.

Note: Because a function returns a value, the function definition states the data type of this value. See more about data types in Chapter 13.

The rules for module identifiers are the same as for variable identifiers (see Section 12.04)

WORKED EXAMPLE 12.11

Drawing a pyramid using modules

The problem is the same as in Worked Example 12.10.

When we want to set up the initial values, we call a procedure, using the following statement:

```
CALL SetValues
```

We can rewrite the top-level solution to our pyramid problem using a procedure for each step, as:

```
CALL SetValues
REPEAT
      CALL OutputSpaces
      CALL OutputSymbols
      CALL AdjustValuesForNextRow
UNTIL NumberOfSymbols > MaxNumberOfSymbols
```

This top-level solution calls four procedures. This means each procedure has to be defined. The procedure definitions are:

```
PROCEDURE SetValues
      INPUT Symbol
      CALL InputMaxNumberOfSymbols // need to ensure it is an odd number
      NumberOfSpaces ← (MaxNumberOfSymbols - 1) / 2
      NumberOfSymbols ← 1
ENDPROCEDURE
PROCEDURE InputMaxNumberOfSymbols
      REPEAT
          INPUT MaxNumberOfSymbols
      UNTIL MaxNumberOfSymbols MOD 2 = 1
ENDPROCEDURE
PROCEDURE OutputSpaces
      FOR Count1 ← 1 TO NumberOfSpaces
```

```
            OUTPUT Space // without moving to next line
        NEXT Count1
ENDPROCEDURE
PROCEDURE OutputSymbols
        FOR Count2 ← 1 TO NumberOfSymbols
            OUTPUT Symbol // without moving to next line
        NEXT Count2
        OUTPUT Newline // move to the next line
ENDPROCEDURE
PROCEDURE AdjustValuesForNextRow
        NumberOfSpaces ← NumberOfSpaces – 1
        NumberOfSymbols ← NumberOfSymbols + 2
ENDPROCEDURE
```

TASK 12.07

Amend your algorithm for Task 12.05 to use modules.

WORKED EXAMPLE 12.12

Drawing a pyramid using modules

The problem is the same as in Worked Example 12.11.

We can rewrite the top-level solution to our pyramid problem using procedures and functions.

```
01    CALL SetValues
02    REPEAT
03          CALL OutputSpaces
04          CALL OutputSymbols
05.1        NumberOfSpaces ← AdjustedNumberOfSpaces
05.2        NumberOfSymbols ← AdjustedNumbeOfSymbols
06    UNTIL NumberOfSymbols > MaxNumberOfSymbols
```

This top-level solution calls three procedures. It also makes use of two functions in lines 05.1 and 05.2.

The procedures and functions have to be defined.

```
PROCEDURE SetValues
    INPUT Symbol
    MaxNumberOfSymbols ← ValidatedMaxNumberOfSymbols
    NumberOfSpaces ← (MaxNumberOfSymbols - 1) / 2
    NumberOfSymbols ← 1
ENDPROCEDURE
FUNCTION ValidatedMaxNumberOfSymbols RETURNS INTEGER
    REPEAT
        INPUT MaxNumberOfSymbols
    UNTIL MaxNumberOfSymbols MOD 2 = 1
    RETURN MaxNumberOfSymbols
ENDFUNCTION
PROCEDURE OutputSpaces
    FOR Count1 ← 1 TO NumberOfSpaces
        OUTPUT Space // without moving to next line
    NEXT Count1
ENDPROCEDURE
```

```
PROCEDURE OutputSymbols
    FOR Count2 ← 1 TO NumberOfSymbols
       OUTPUT Symbol // without moving to next line
    NEXT Count2
    OUTPUT Newline // move to the next line
ENDPROCEDURE
FUNCTION AdjustedNumberOfSpaces RETURNS INTEGER
    NumberOfSpaces ← NumberOfSpaces – 1
    RETURN NumberOfSpaces
ENDFUNCTION
FUNCTION AdjustedNumberOfSymbols RETURNS INTEGER
    NumberOfSymbols ← NumberOfSymbols + 2
    RETURN NumberOfSymbols
ENDFUNCTION
```

Note that procedure `SetValues` uses a function `ValidatedMaxNumberOfSymbols`.

One benefit of using modules is that individual modules can be reused in other solutions. Therefore, modules should be designed to be self-contained. That means they should not rely on external variables. All variables that are required by a module should be passed to it using parameters. To illustrate this, look at Worked Example 12.13

WORKED EXAMPLE 12.13

Drawing a pyramid using modules and parameters

The problem is the same as in Worked Example 12.12.

```
01   CALL SetValues(Symbol, MaxNumberOfSymbols, NumberOfSpaces, NumberOfSymbols)
02   REPEAT
03        CALL OutputSpaces(NumberOfSpaces)
04        CALL OutputSymbols(NumberOfSymbols, Symbol)
05.1      NumberOfSpaces ← AdjustedNumberOfSpaces(NumberOfSpaces)
05.2      NumberOfSymbols ← AdjustedNumbeOfSymbols(NumberOfSymbols)
06   UNTIL NumberOfSymbols > MaxNumberOfSymbols
```

Module definitions:

```
PROCEDURE SetValues(Symbol, MaxNumberOfSymbols, NumberOfSpaces, NumberOfSymbols)
     INPUT Symbol
     MaxNumberOfSymbols ← ValidatedMaxNumberOfSymbols
     NumberOfSpaces ← (MaxNumberOfSymbols - 1) / 2
     NumberOfSymbols ← 1
ENDPROCEDURE
FUNCTION ValidatedMaxNumberOfSymbols RETURNS INTEGER
     REPEAT
         INPUT MaxNumberOfSymbols
     UNTIL MaxNumberOfSymbols MOD 2 = 1
     RETURN MaxNumberOfSymbols
ENDFUNCTION
PROCEDURE OutputSpaces(NumberOfSpaces)
     FOR Count1 ← 1 TO NumberOfSpaces
         OUTPUT Space // without moving to next line
     NEXT Count1
ENDPROCEDURE
PROCEDURE OutputSymbols(NumberOfSymbols, Symbol)
```

```
    FOR Count2 ← 1 TO NumberOfSymbols
        OUTPUT Symbol // without moving to next line
    NEXT Count2
    OUTPUT Newline // move to the next line
ENDPROCEDURE
FUNCTION AdjustedNumberOfSpaces(NumberOfSpaces) RETURNS INTEGER
    NumberOfSpaces ← NumberOfSpaces – 1
    RETURN NumberOfSpaces
ENDFUNCTION
FUNCTION AdjustedNumbeOfSymbols(NumberOfSymbols) RETURNS INTEGER
    NumberOfSymbols ← NumberOfSymbols + 2
    RETURN NumberOfSymbols
ENDFUNCTION
```

Note that the procedure `OutputSpaces` uses a variable, `Count1`, which is used only within the module. Similarly, `OutputSymbols` uses variable `Count2` only within the module. We call such a variable a **local variable** (see Chapter 14, Section 14.09). A variable available to all modules is known as a **global variable** (see Chapter 14, Section 14.09).

TIP

Good design uses local variables as it makes modules independent and re-usable.

Reflection Point:

Can you think of other problems and use decomposition to break them down into basic constructs, input and output statements?

Summary

- Abstraction involves filtering out information that is not needed to solve the problem.
- Decomposition is breaking down problems into sub-problems, leading to the concept of a program module.
- An algorithm is a sequence of steps that can be carried out to solve a problem.
- Algorithms are expressed using the four basic constructs of assignment, sequence, selection and repetition.
- Algorithms can be documented using pseudocode.
- Stepwise refinement: breaking down the steps of an outline solution into smaller and smaller steps.
- Logic statements use the relational operators =, <, >, <>, <= and >= and the logic operators AND, OR and NOT.
- Selection constructs and conditional loops use conditions to determine the steps to be followed.

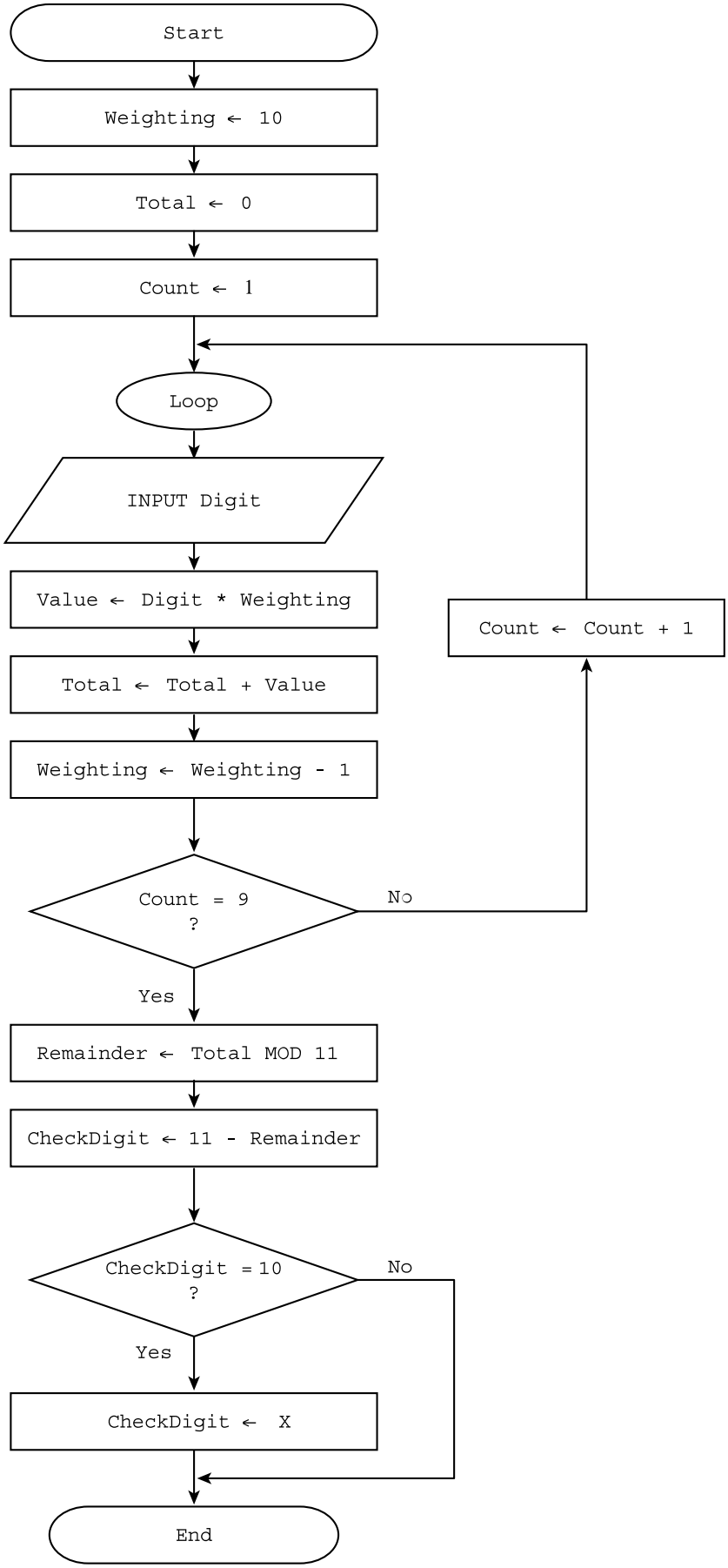
Exam-style Questions

1 The Modulo-11 method of calculating a check digit for a sequence of nine digits is as follows:

Each digit in the sequence is given a weight depending on its position in the sequence. The leftmost digit has a weight of 10. The next digit to the right has a weight of 9, the next one 8 and so on. Values are calculated by multiplying each digit by its weight. These values are added together and the sum is divided by 11. The remainder from this division is subtracted from 11 and this value is the check digit. If this value is 10, then the check digit is X. Note that `x MOD y` gives the remainder from the division of `x` by `y`.

The flowchart shows the algorithm for calculating the Modulo-11 check digit.

Write pseudocode from the flowchart.



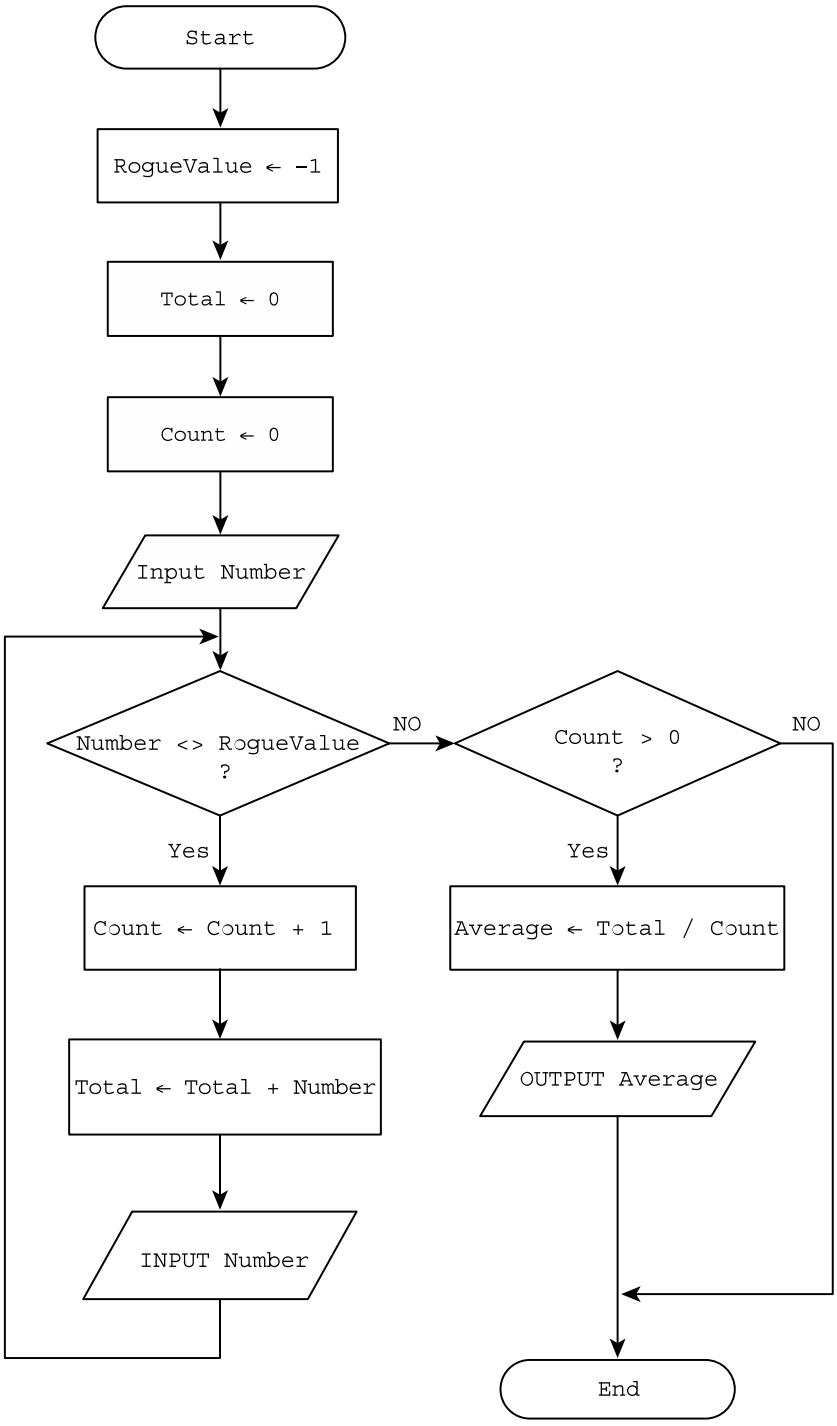
[9]

2 Write pseudocode for the following problem given in structured English.

```
REPEAT the following UNTIL the number input is zero
INPUT a number
Check whether number is positive or negative
Increment positive number count if the number is positive
```

```
Increment negative number count if the number is negative
```
[7]

3 Write pseudocode from the given flowchart. Use a `WHILE` loop.



[8]