

Programming AS

Read online at pastopic.com/cambridge/computer-science-9618/notes/programming

Last checked 5 October 2026

Read first: [Algorithm design and problem-solving](#), [Data types and structures](#)

© 2026 Pastopic. Free for personal study. Please share the link, not the file.

What you need to know

This topic is examined in Paper 2, in almost every question. By the end of it you should be able to:

1. **Turn a design into pseudocode:** write pseudocode from a program flowchart or from structured English.
2. **Basic statements:** declare and initialise **constants**, declare **variables**, **assign** values, and write expressions with the arithmetic operators, the logic operators (AND, OR, NOT), input from the keyboard (INPUT) and output to the screen (OUTPUT).
3. **Built-in functions and library routines:** use the functions in the Paper 2 insert (string, numeric and date functions). Any function not in the pseudocode guide is described in the question.
4. **Selection:** write an IF statement with an ELSE clause, **nested** IF statements, and a CASE structure.
5. **Iteration:** write a **count-controlled** loop (FOR), a **post-condition** loop (REPEAT ... UNTIL) and a **pre-condition** loop (WHILE ... ENDWHILE), and **justify** why one loop suits a problem better than the others.
6. **Procedures:** define and use a procedure, and explain where in an algorithm a procedure is the right choice.
7. **Parameters:** use none, one or more parameters, passed **by value** or **by reference**.
8. **Functions:** define and use a function, and explain where a function is the right choice: it returns a value, so the call is used inside an expression and the returned value takes the call's place.
9. **Terminology:** procedure or function **header**, **interface**, **parameter**, **argument** and **return value**.
10. **Efficiency:** write pseudocode that does no unnecessary work.

Data types, arrays, records and text files are taught in the Data types and structures note; flowchart symbols and the basic constructs first appear in the Algorithm design note. This note shows how to use them all to write, read and fix programs.

Understand it

Statements: constants, variables, assignment, input and output

```
CONSTANT MinAge = 16           // a constant: its value never changes while the program runs
DECLARE Age : INTEGER          // a variable: a named store whose value can change
DECLARE Name : STRING
INPUT Name                     // from the keyboard
INPUT Age
Age ← Age + 1                  // assignment: work out the right side, store it on the left
OUTPUT "Next year ", Name, " will be ", Age // to the screen; commas separate the items
```

A **constant** is a value given a name once, at the start. Use one for any fixed value that appears in several places (a minimum age, a maximum number of items): the name explains the number, the value can't be changed by mistake while the program runs, and if it ever changes you edit one line instead of hunting for every copy.

Operators (all in the insert):

- arithmetic: +, −, *, / (always gives a REAL), DIV (the whole-number part of a division) and MOD (the remainder): 47 DIV 6 is 7 and 47 MOD 6 is 5;
- comparison: =, <>, <, >, <=, >=, which give TRUE or FALSE. Comparing strings is case sensitive: "Kiwi" = "kiwi" is FALSE;
- logic: AND, OR, NOT;
- & joins (concatenates) strings or characters: "Wai" & "kato" is "Waikato".

= compares; ← assigns. Writing `IF Count ← 10 THEN` is a syntax error.

Built-in functions and library routines

A **built-in function** is part of the language: you call it and use the value it gives back. The Paper 2 insert lists every function you may use, with its parameters and return type. Use only these (or ones defined in the question), never ones from Python or another language.

Function	Gives	Example
<code>LENGTH(S)</code>	INTEGER: number of characters	<code>LENGTH("Rua 4")</code> is 5
<code>LEFT(S, n), RIGHT(S, n)</code>	STRING: first or last n characters	<code>RIGHT("Taupo", 2)</code> is "po"
<code>MID(S, x, n)</code>	STRING: n characters starting at position x	<code>MID("Taupo", 2, 3)</code> is "aup"
<code>TO_UPPER(S), TO_LOWER(S)</code>	the same type, with the letters changed	<code>TO_UPPER("b7c")</code> is "B7C"
<code>NUM_TO_STR(N), STR_TO_NUM(S)</code>	converts a number to a string or back	<code>STR_TO_NUM("6.5")</code> is 6.5
<code>IS_NUM(S)</code>	BOOLEAN: TRUE if S is a valid number	<code>IS_NUM("12a")</code> is FALSE
<code>ASC(C), CHR(N)</code>	a character's ASCII code, or the character	<code>ASC('a')</code> is 97, <code>CHR(90)</code> is 'Z'
<code>INT(X)</code>	INTEGER: the whole-number part	<code>INT(8.97)</code> is 8
<code>RAND(N)</code>	REAL from 0 up to, but not including, N	<code>RAND(10)</code> could be 3.0751
<code>DAY(D), MONTH(D), YEAR(D)</code>	INTEGER parts of a date	<code>MONTH(17/09/2026)</code> is 9
<code>DAYINDEX(D)</code>	INTEGER day of the week, Sunday = 1, Monday = 2, ... Saturday = 7	
<code>SETDATE(Day, Month, Year)</code>	DATE built from three integers	<code>SETDATE(2, 1, 2027)</code>
<code>TODAY()</code>	DATE: the current date	

Every function needs arguments of the right **type**. `LENGTH(472)` is an error, because 472 is not a string; `Score: " & 85` is an error, because `&` joins strings, not numbers. Convert first: `LENGTH(NUM_TO_STR(472))` is 3.

A **library routine** is a ready-made, tested subroutine stored in a program library, which a programmer can call instead of writing that code. Built-in functions are one example; a library might also provide encryption or sorting routines. Benefits:

- it has already been **tested**, so it is less likely to contain errors;
- it is **ready to use**, which saves development time;
- it may do something the programmer could not write themselves (complex maths, encryption to a recognised standard);
- it is **maintained** by its authors: improvements and fixes arrive without the programmer doing anything.

Selection: IF and CASE

```
IF Speed > 50 THEN
    OUTPUT "Too fast"
ELSE
    OUTPUT "OK"
ENDIF
```

A **nested IF** is an IF inside another IF (or inside its ELSE). It tests a second condition only when the first has a certain result:

```
IF Age >= MinAge THEN
    IF HasPermit = TRUE THEN
        OUTPUT "May drive"
    ELSE
        OUTPUT "Needs a permit"
    ENDIF
ELSE
    OUTPUT "Too young"
ENDIF
```

A **CASE** structure chooses one of several branches using the value of one variable or expression:

```
CASE OF Rating
  1      : OUTPUT "Poor"
  2 TO 3 : OUTPUT "Fair"
  4      : OUTPUT "Good"
  5      : OUTPUT "Excellent"
  OTHERWISE : OUTPUT "Invalid rating"
ENDCASE
```

The clauses are tested **in order** and only the **first** match runs; OTHERWISE runs if none match. A clause can be a value, a range (2 TO 3), or on some papers a comparison (< 50 :). Because only the first match runs, a clause whose values are all caught by an earlier clause can never run: in a CASE that starts < 50 : and then has < 20 :, the second clause is never reached.

Use CASE when one variable is compared with several fixed values (a menu choice, a code letter): it is shorter and easier to read than a chain of IFs. Use IF when the conditions involve different variables or AND/OR. Any CASE can be rewritten as nested IFs:

```
IF Rating = 1 THEN
  OUTPUT "Poor"
ELSE
  IF Rating >= 2 AND Rating <= 3 THEN
    OUTPUT "Fair"
  ELSE
    ...
```

Iteration: three kinds of loop

```
FOR Lap ← 1 TO 8                                // count-controlled
  OUTPUT Lap
NEXT Lap

FOR Countdown ← 10 TO 0 STEP -2                  // 10, 8, 6, 4, 2, 0
  OUTPUT Countdown
NEXT Countdown

WHILE Balance < Target                            // pre-condition: tested before each pass
  Balance ← Balance * 1.05
  Years ← Years + 1
ENDWHILE

REPEAT                                           // post-condition: tested after each pass
  INPUT Choice
UNTIL Choice >= 1 AND Choice <= 4
```

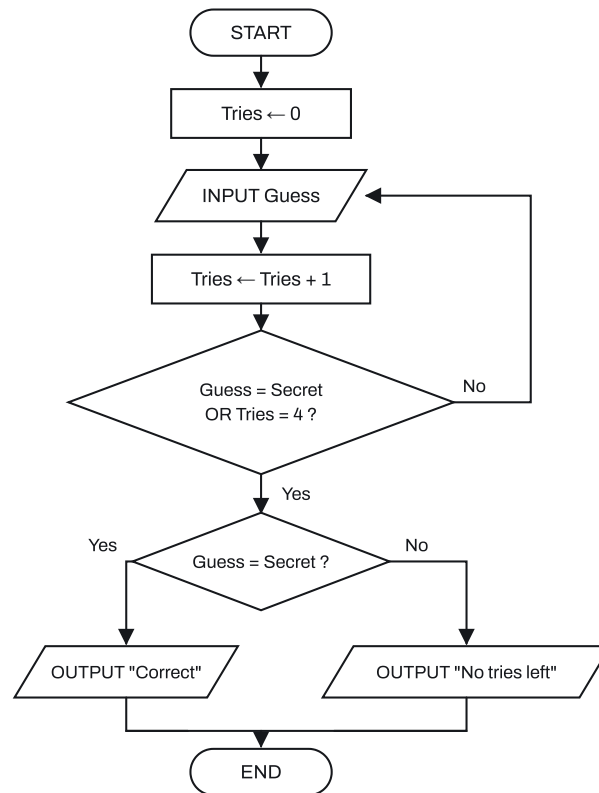
Loop	When the test happens	Runs at least once?	Use it when
FOR ... NEXT (count-controlled)	it counts a fixed number of passes	yes, unless the range is empty	the number of passes is known before the loop starts: every element of an array, every character of a string
WHILE ... ENDFILE (pre-condition)	before each pass	no: it may run 0 times	it must stop as soon as something happens, and possibly before the first pass: reading a file that might be empty, searching until found
REPEAT ... UNTIL (post-condition)	after each pass	yes	the body must run once before there is anything to test: input that must be validated, a game turn

Two facts the examiners often test: a WHILE loop **continues while** its condition is TRUE, and a REPEAT loop **stops when** its condition becomes TRUE, so the two conditions are opposites. And a FOR loop always does all its passes, so a search written with FOR keeps comparing after the item is found, which is inefficient. A conditional loop with a flag (WHILE Index <= 100 AND NOT Found), or a RETURN inside the loop of a function, stops at once.

From a flowchart or structured English to pseudocode

Convert box by box, keeping the order:

- a process box becomes an assignment; a parallelogram becomes INPUT or OUTPUT;
- a decision that skips **forward** becomes an IF;
- a decision whose arrow goes **back up** is a loop. If the test comes **after** the boxes it repeats, it is a REPEAT ... UNTIL (the condition that leaves the loop goes after UNTIL); if the test comes **before** them, it is a WHILE (the condition that stays in the loop goes after WHILE).



The first decision comes after the input and the count, and its "No" arrow goes back up, so it is a post-condition loop. The second decision only skips forward, so it is an IF:

```

Tries ← 0
REPEAT
    INPUT Guess
    Tries ← Tries + 1
UNTIL Guess = Secret OR Tries = 4
IF Guess = Secret THEN
    OUTPUT "Correct"
ELSE
    OUTPUT "No tries left"
ENDIF
  
```

Structured English converts the same way: "repeat ... until ..." is a REPEAT loop, "for each ..." is usually a FOR loop, "if ... otherwise ..." is an IF with an ELSE.

Procedures and functions

A **subroutine** is a named block of code that does one task and can be called from anywhere in the program. There are two kinds.

A **procedure** does a task. It is called with a CALL statement, as a statement on its own:

```

PROCEDURE ShowBar(Length : INTEGER, Symbol : CHAR)
    DECLARE Count : INTEGER
  
```

```

DECLARE Bar : STRING
Bar ← ""
FOR Count ← 1 TO Length
    Bar ← Bar & Symbol
NEXT Count
OUTPUT Bar
ENDPROCEDURE

```

```
CALL ShowBar(12, '#')
```

A **function** does a task and **returns one value**. Its header says the type of that value, it must end by executing a **RETURN**, and it is called **inside an expression**: when it finishes, the returned value takes the place of the call.

```

FUNCTION Larger(A : INTEGER, B : INTEGER) RETURNS INTEGER
    IF A > B THEN
        RETURN A
    ENDIF
    RETURN B
ENDFUNCTION

```

```

Best ← Larger(Score1, Score2) + Bonus
IF Larger(Height, Width) > 600 THEN ...

```

A function is never called with **CALL**, and its result must be used (assigned, output, compared). **OUTPUT** inside a function is not the same as **RETURN**: output goes to the screen, and the calling code receives nothing.

When to use a subroutine:

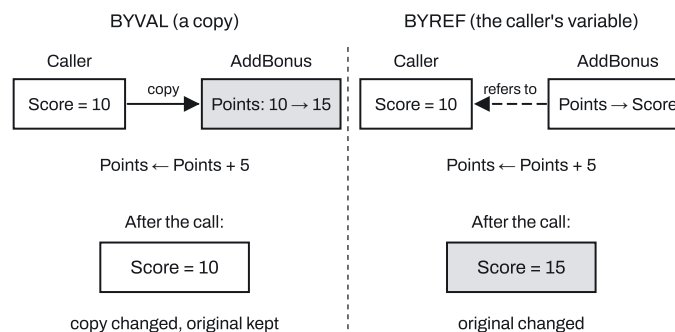
- the same task is needed in **several places**: write it once and call it each time;
- a part of the algorithm does one **specific, separate task** (decomposition turns each such task into a module);
- each module can be written, **tested and debugged** on its own, by different programmers, and later **maintained** in one place;
- the main program becomes shorter and easier to understand.

Procedure or function? Use a **function** when the task produces **one value** that the calling code will use in a calculation, a condition or an assignment (a total, a valid/invalid flag, a converted string). Use a **procedure** when the task produces no value (it outputs, writes a file, or updates global data), or when it must hand back **more than one** value, which it does through **BYREF** parameters.

Parameters: by value and by reference

A **parameter** is a variable named in the header, which receives a value when the subroutine is called. An **argument** is the actual value or variable given in the call. In `CALL ShowBar(12, '#')` the arguments are 12 and '#'; the parameters are `Length` and `Symbol`.

- **By value (BYVAL, the default)**: the subroutine gets a **copy** of the argument. Changes to the parameter inside the subroutine do not affect the caller's variable.
- **By reference (BYREF)**: the parameter refers to the caller's **own variable** (its address is passed). Any change made inside the subroutine changes the caller's variable, so the new value is still there after the call.



```
PROCEDURE AddBonus(BYREF Points : INTEGER)
    Points ← Points + 5
ENDPROCEDURE
```

With `Score ← 10` then `CALL AddBonus (Score)`, `Score` is 15 afterwards. Without `BYREF`, `Score` would still be 10. Write `BYREF` in front of each parameter that must pass a change back, for example `PROCEDURE Swap(BYREF First : INTEGER, BYREF Second : INTEGER)`.

Use `BYREF` when a procedure must give **values back** to the caller (two counts, an updated date). In a structure chart, a value that goes back up to the calling module is a `BYREF` parameter of a procedure, or the return value of a function if it is the only one.

Terms:

- **Header:** the first line, giving the name, the parameters with their types (and `BYREF` if used) and, for a function, `RETURNS` and the return type:
`FUNCTION Larger(A : INTEGER, B : INTEGER) RETURNS INTEGER.`
- **Interface:** how the rest of the program uses the subroutine: its name, the number, order and types of its parameters (and how each is passed), and the type of its return value. Callers need only the interface, not the code inside.
- **Return value:** the single value a function sends back with `RETURN`.

Local and global variables

A **local** variable is declared inside a subroutine. It exists only while that subroutine runs and can be used only inside it; its value is lost when the subroutine ends. A **global** variable is declared in the main program and can be used anywhere, for the whole run.

Local variables are better wherever they will do:

- the same name can be used in other subroutines without clashing;
- a subroutine can't accidentally change data used elsewhere, so each module is self-contained and easier to test and reuse;
- their memory is freed when the subroutine ends.

A value that must be kept **between calls** (a running total across many calls) can't be a local variable, because a local starts again at every call: use a global, or pass the value in and back as a parameter.

Writing efficient and readable pseudocode

Efficient pseudocode does no work it doesn't need:

- stop a loop as soon as its job is done (found the item, reached a limit);
- don't call the same function again and again with the same arguments inside a loop: call it once before the loop and keep the result in a variable;
- don't test a condition that an earlier test has already settled (an `ELSE` instead of a second `IF` with the opposite condition);
- skip unused elements quickly, and on sorted data stop once you've passed where the item would be.

Readable pseudocode (good programming practice) uses meaningful identifiers (`StockLevel`, not `S` or `X1`), comments to explain each part, indentation inside every `IF`, `CASE`, loop and subroutine, blank lines between sections, constants instead of unexplained numbers, local variables, and an array instead of many numbered variables. All of these make a program easier to understand, debug and maintain.

Dry running with a trace table

A **dry run** means following code line by line by hand. Record it in a **trace table**: one column per variable (and one for output), and a new value written each time a variable changes. Trace this:

```
Total ← 0
Step ← 1
WHILE Total < 12
    CASE OF Step MOD 3
        0 : Total ← Total + 1
        1 : Total ← Total + Step
        OTHERWISE : Total ← Total * 2
    ENDCASE
    Step ← Step + 1
```

ENDWHILE

OUTPUT Total, " ", Step

Step (at the start of the pass)	Step MOD 3	Clause	Total after
1	1	Total + Step	1
2	2	OTHERWISE	2
3	0	Total + 1	3
4	1	Total + Step	7
5	2	OTHERWISE	14

After the fifth pass Step is 6 and Total is 14, which is not less than 12, so the loop ends and the output is 14 6. Check the loop condition each time **before** a pass of a WHILE (after a pass of a REPEAT).

Errors

- A **syntax error** breaks the rules of the language, so the program can't be translated: IF Total ← 100 THEN, ENDIF missing, a FOR loop closed with ENDWHILE, a misspelt keyword or a function that doesn't exist.
- A **logic error** lets the program run but gives the wrong result: > instead of >=, a counter never reset, a percentage not divided by 100, CASE clauses in an order that makes one unreachable.
- A **run-time error** happens while the program runs and makes it crash or hang: dividing by zero, an array index outside the bounds, STR_TO_NUM on a string that isn't a number, or a loop whose condition never becomes true (an infinite loop, so the program stops responding).

Key facts and formulas

There is no formula list for 9618. The built-in functions and operators are printed in the Paper 2 insert (marked Given below); the statement forms are not, so learn them.

Formula	Status
String functions: LENGTH(S), LEFT(S, n), RIGHT(S, n), MID(S, x, n), TO_UPPER, TO_LOWER, NUM_TO_STR, STR_TO_NUM, IS_NUM, ASC, CHR	Given
Numeric functions: INT(X) (whole-number part), RAND(N) (REAL from 0 up to but not including N)	Given
Date functions: DAY, MONTH, YEAR, DAYINDEX (Sunday = 1), SETDATE(Day, Month, Year), TODAY()	Given
Operators: & joins strings; DIV gives the quotient and MOD the remainder; AND, OR, NOT; =, <>, <, >, <=, >=	Given
Constant and variable: CONSTANT Name = Value; DECLARE Name : Type; assignment Name ← Expression	Learn it
Selection: IF ... THEN ... ELSE ... ENDIF; CASE OF Var ... Value : ... OTHERWISE : ... ENDCASE (first matching clause runs)	Learn it
Loops: FOR Var ← A TO B [STEP S] ... NEXT Var; WHILE Cond ... ENDWHILE; REPEAT ... UNTIL Cond	Learn it
Procedure: PROCEDURE Name(P : Type, ...) ... ENDPROCEDURE, used with CALL Name(...)	Learn it
Function: FUNCTION Name(P : Type, ...) RETURNS Type ... RETURN Value ... ENDFUNCTION, used inside an expression	Learn it
BYVAL (default) passes a copy; BYREF passes the caller's variable, so changes are kept after the call	Learn it
Random whole number from 1 to N: INT(RAND(N)) + 1	Learn it
Last digit of a whole number: Num MOD 10; the number without its last digit: Num DIV 10	Learn it

Formula	Status
Any substring from LEFT and RIGHT: $MID(S, x, n) = RIGHT(LEFT(S, x + n - 1), n)$	Learn it

How to do it

In the Paper 2 questions we have tagged for this topic (124 questions, 2021–2025; 5 of them are the same question on two papers), the types came up this often. 7 of the 124 touch this topic only in parts that belong to other topics (structure charts, abstraction, testing), so they are not in the table. 63 of the other 117 questions mix types, so the counts add up to more than 117.

Question type	Questions
Write a function or procedure that processes a string	29
Write a module that searches, counts or updates an array	22
Write a module that reads or writes a text file	18
Give the data type of each variable	18
Write a module with numbers, random values, input or dates	16
Evaluate or complete expressions using built-in functions	15
Write headers and explain parameters, BYVAL and BYREF, functions and procedures	15
Identify the constructs in code or name a construct an algorithm needs	11
Rewrite given code	11
Explain why to use subroutines, library routines and local variables	11
Find and explain errors in code	10
Dry-run code or answer questions about it	10
Choose and justify a loop structure	9
Write a single statement or condition	7
Good programming practice	7
Replace literal values with constants	3

Every "write pseudocode" answer is marked point by point: the header and ending, initialising before the loop, the right loop, the key test inside it, the update, and the result returned or output. The question's bullet points usually match the mark points, so tick each one off.

1. Write a function or procedure that processes a string (29 questions)

1. **Header** with the exact name, each parameter with its type, and for a function RETURNS with the type; finish with ENDFUNCTION or ENDPROCEDURE.
2. **Declare and initialise** before the loop: a counter to 0, a new string to " ".
3. **Loop** FOR Index ← 1 TO LENGTH(Text) to visit every character.
4. **Take one character** with MID(Text, Index, 1), not LEFT (which always starts at the beginning).
5. **Test it and act:** count it, or join it (or a replacement) to the new string with &.
6. **After the loop**, RETURN the result (or OUTPUT it for a procedure that is asked to output).

Variations you will meet:

- **Count a character, or count two kinds** (vowels and the rest): test with IF, or CASE for several letters. Give each count its own test; an ELSE that counts "everything else" may also count spaces and digits.
- **Replace characters, remove characters, reverse:** build a new string. To reverse, loop FOR Index ← LENGTH(Text) TO 1 STEP -1, or put each character in front: NewText ← ThisChar & NewText.
- **Ignore case:** convert both strings with TO_UPPER (or TO_LOWER) before comparing.

- **Compare with the start or end of another string:** first check the second string is long enough, then compare with `LEFT ( Long , LENGTH ( Short ) )` Or `RIGHT ( Long , LENGTH ( Short ) )`.
- **Leading spaces, a comment marker, or text after a separator:** find the position with a `WHILE` loop that stops at the first character that is not a space (or that is the separator), then cut with `RIGHT` or `LEFT`. Guard the loop so it can't run past the end: `WHILE Pos <= LENGTH ( Text ) AND MID ( Text , Pos , 1 ) = ' '`.
- **Split a list at separators** (such as "4,19,250"): build up each item character by character; when a separator (or the end of the string) is reached, use the item (`STR_TO_NUM` if it is a number) and reset it to "".
- **Fixed-format strings** (an ID of 2 letters then 4 digits): check `LENGTH` first, then test parts with `MID` and `IS_NUM`, or call a helper function the question gives you. Use the helper's return value: `IF IsLetters ( LEFT ( Code , 2 ) ) = TRUE THEN`.
- **Mask part of a string:** keep the last few characters with `RIGHT`, and build the rest from a repeated character in a loop.
- **Write your own substring function** from `LEFT` and `RIGHT` only: `RIGHT ( LEFT ( S , x + n - 1 ) , n )`.

2. Write a module that searches, counts or updates an array (22 questions)

These are the long questions at the end of Paper 2: a global array (often of records), with unused elements marked by a special value.

1. **Header** with the given name and parameters; declare the local index and any counters.
2. **Choose the loop.** Count or total over the whole array: `FOR Index ← 1 TO Size`. Find one item or stop at a limit ("efficient"): `WHILE Index <= Size AND NOT Found`, or a `RETURN` inside a `FOR` loop of a function.
3. **Reach the data** with the index and dot notation: `Bookings [ Index ] . Paid`.
4. **Skip unused elements** (key field "", or a value such as -1 or 9999), then test the fields the question names, joined with `AND`.
5. **Act:** count, total, set a field, or fill the first unused element (set every field).
6. **After the loop,** give **both** outcomes exactly once: found or not found, `TRUE` or `FALSE`, the count with a message.

Variations you will meet:

- **Count up to a limit** ("may borrow at most 3"): stop the loop as soon as the count reaches the limit, then output the right message.
- **Find and update in one pass:** when the element is found, check its state first (already returned? already full?), then change it and return `TRUE`; return `FALSE` if it was found in the wrong state **or** not found at all.
- **Sorted data:** stop once you pass the place where the item would be, or use a binary search, or calculate the index directly when the keys run in sequence (key 5001 is at index 1, so `Index ← Key - 5000`; check the key is in range first).
- **Average:** keep a sum and a count in the loop, divide **after** the loop, declare the average as `REAL`.
- **Outline changes for a new requirement** (such as fines for late returns): say which **data** must be added (a new field in the record type, such as the due date, or a category that decides the loan length) and how the **program uses it** (calculate the due date from the loan date, compare it with `TODAY ( )`).

3. Write a module that reads or writes a text file (18 questions)

1. **Open** each file in the right mode: `READ`, `WRITE` (new file, erases an old one) or `APPEND` (adds to the end). A file name held in a parameter is written **without** quotes: `OPENFILE FileName FOR READ`.
2. **Loop** `WHILE NOT EOF ( FileName )`: a pre-condition loop, because the file might be empty.
3. **Read a line** inside the loop: `READFILE FileName , Line`.
4. **Process it:** cut out the part you need with `LEFT`, `MID` or `RIGHT`, convert with `STR_TO_NUM`, test it, count it, or pass it to a function the question gives you.
5. **Write** a line with `WRITEFILE` (convert numbers to strings first).
6. **After the loop,** close every file you opened and output the count or result with a message.

Variations you will meet:

- **Copy some lines to a new file:** two files open at once, one `READ` and one `WRITE`.
- **Totals for groups of lines** (all lines for the same day, when the file is in order): keep the current key; when a line's key differs, output the total for the old key, then start again with the new key and a count of 1. After the loop, output the last group too.
- **Many files with numbered names:** build each name with `& ( "Log" & NUM_TO_STR ( Num ) & ".txt" )` in a `FOR` loop, and open each one for `WRITE`.

- **Fill an array from a file:** stop at the end of the file **or** when the array is full.

The design questions about files (why use a file, separators, one field per line) are in the Data types and structures note.

4. Give the data type of each variable (18 questions)

One mark per row. Money, averages and measurements are REAL; counts are INTEGER; one character is CHAR; text and codes are STRING; yes/no is BOOLEAN; a date without quotes is DATE. An identifier table also wants a meaningful name (`DateJoined`, `Fee`). The full rules are in the Data types and structures note; here the point is to write the right type in every `DECLARE` and every header.

5. Write a module with numbers, random values, input or dates (16 questions)

1. **Header and declarations**, with REAL for anything that can have a fraction (an average).
2. **Initialise** totals and counters before the loop.
3. **The loop:** FOR for a known number of values or rolls, WHILE or REPEAT for "until".
4. **The calculation** inside it; **return or output** the result after it.

Variations you will meet:

- **Digits of a number:** `Num MOD 10` is the last digit and `Num DIV 10` removes it; repeat WHILE `Num > 0`. Or convert with `NUM_TO_STR` and take each character with `MID`, converting back with `STR_TO_NUM`. To add a digit to the end of a number: `Num * 10 + Digit`.
- **Random values:** `INT(RAND(N)) + 1` gives a whole number from 1 to N (`RAND` alone gives a REAL, and `RAND(N)` never reaches N). A random capital letter: `CHR(65 + INT(RAND(26)))`.
- **Input until a condition:** a REPEAT loop (or a WHILE with an input before it and another at the end of the body). Ignore values until a trigger, then total: use a BOOLEAN flag.
- **Dates:** use the date functions, not string functions, on a DATE. "Same month as today": `MONTH(D) = MONTH(TODAY())`. "A Monday": `DAYINDEX(D) = 2`. A date from parts: `SETDATE(Day, Month, Year)`.
- **Clock values:** add 1 to the seconds; if they reach 60, set them to 0 and add 1 to the minutes; then the same for minutes into hours, and hours back to 0 after 23.
- **Patterns:** nested FOR loops, the outer for each row, the inner building that row's string, `OUTPUT` after the inner loop.
- **Products such as a factorial:** start the running product at 1, not 0.
- **A procedure that must change the caller's variables** (update points and a date): BYREF parameters.

6. Evaluate or complete expressions using built-in functions (15 questions)

1. **Work from the inside out**, one function at a time, using the values given.
2. **Count positions from 1** for `MID`.
3. **Write the result in its type's form:** strings in double quotes ("12", not 12), characters in single quotes, TRUE or FALSE for conditions, numbers plain.
4. **Write ERROR** when a type is wrong: a number joined with `&`, a string added to a number, `LENGTH` of a number.

Variation: complete the expression to give a stated value. Work backwards: 'D' from 68 needs `CHR`; FALSE from TRUE needs `NOT`; a number from a string needs `STR_TO_NUM`.

7. Write headers and explain parameters, BYVAL and BYREF, functions and procedures (15 questions)

1. **Function or procedure?** One value comes back to be used → function. Nothing comes back, or several values come back → procedure (the values come back through BYREF).
2. **Write the header:** name, each parameter with `: Type`, BYREF in front of any parameter whose change must be kept, and `RETURNS Type` for a function.
3. From a **structure chart:** a value passed down is a parameter; a value passed back up is the return value (if it is the only one, from a function) or a BYREF parameter; a value passed both ways is BYREF.

Variations you will meet:

- **Explain the difference:** a function returns a single value and is used in an expression; a procedure does not return a value and is called with `CALL`.
- **Explain BYVAL and BYREF:** by value passes a copy, so the original can't be changed; by reference passes the variable's address, so changes made in the subroutine are made to the original.
- **Describe the terms** in a given header: the name; each parameter (the value passed in) and its type; the return type (the type of the value returned).
- **Turn a procedure into a function:** change the header to `FUNCTION ... RETURNS Type`, remove the `BYREF` parameter that carried the result, add `RETURN`, change `ENDPROCEDURE` to `ENDFUNCTION`, and change the call from `CALL Name (...)` to an assignment such as `Result ← Name (...)`.

8. Identify the constructs in code or name a construct an algorithm needs (11 questions)

Tick every construct that appears in each line or block:

- **Assignment:** any `←` (including `x ← Func ( )`, which is also a subroutine call).
- **Selection:** `IF`, `ELSE`, `CASE OF`, a clause such as `OTHERWISE` or `'B' : .`
- **Iteration:** `FOR`, `WHILE`, `REPEAT`, `UNTIL`, `NEXT`.
- **Subroutine:** `CALL`, or a function name with brackets used in an expression or condition (`WHILE IsEmpty ( ) = FALSE`). Built-in functions count as function calls.
- **Input/output:** `INPUT`, `OUTPUT`, `READFILE`, `WRITEFILE`.

To **name a construct an algorithm needs**, give it and its use in this algorithm: selection, to test whether each element is blank before writing it; iteration, to repeat for each element.

9. Rewrite given code (11 questions)

- **CASE to IF:** one `IF` per clause, nested in each `ELSE`; `OTHERWISE` becomes the last `ELSE`. Clauses that do the same thing can share one condition with `OR`.
- **Merge CASE clauses** that differ only by a number equal to the case value: clauses `3 : Pos ← Pos + 3` and `4 : Pos ← Pos + 4` become `3 TO 4 : Pos ← Pos + Move`, using the case variable `Move` itself.
- **Flatten nested IFs:** write each route through the nest as one `IF` whose condition joins the tests with `AND` (and `NOT` for the `ELSE` branches), then simplify.
- **Simplify conditions:** `NOT ( A OR B )` is `NOT A AND NOT B`; `NOT ( A AND B )` is `NOT A OR NOT B`; `X > 5 OR X = 5` is `X >= 5`.
- **Flowchart to pseudocode:** as in "Understand it": a decision that goes back up is a loop.
- **Make it efficient:** move a repeated function call (`LENGTH ( Text )` in a `WHILE` condition, or a lookup that doesn't change) to before the loop and store the result; change a `FOR` search into a conditional loop.

10. Explain why to use subroutines, library routines and local variables (11 questions)

Give short points that fit the scenario:

- **Subroutines:** the task is repeated or needed in several places; it is a distinct task; the code is written once, so the program is shorter; easier to test, debug and maintain; can be reused.
- **Library routines:** already tested so free of errors; available now, so development is faster; can do things the programmer couldn't; updated by their authors; follow recognised standards.
- **Local variables:** exist only in their module (a global is visible everywhere for the whole run); the same name can be reused elsewhere; a module can't accidentally change other data; memory is freed after the module ends. A value needed between calls can't be local.

11. Find and explain errors in code (10 questions)

1. **Read each statement against the insert:** are the arguments the right type and number? Is it a real function? Is `←` used only for assignment?
2. **Check each structure is closed** by the right keyword (`ENDIF`, `ENDCASE`, `NEXT Var`, `ENDWHILE`, `UNTIL`).
3. **Say what is wrong and how to fix it**, or "no error". For a run-time error, give the input that causes it (a divisor of 0, an index of 0 for `ARRAY[ 1 : 100 ]`).

4. For a **logic error**, compare the code with what it should do on a simple test value: does a percentage get divided by 100? Can every CASE clause be reached? Does the loop end?

12. Dry-run code or answer questions about it (10 questions)

1. **Draw the table** with a column for each variable and output, in the order they appear.
2. **Follow one line at a time**, writing a new value only when it changes. Test the WHILE condition before each pass.
3. **For CASE**, find the first clause that matches.

Questions about given code ask what a part of it is for. Take this fragment, which looks through 40 race times:

```
Fastest ← 0
FOR Runner ← 1 TO 40
    IF Times[Runner] < 60.0 THEN
        IF Fastest = 0 THEN
            Fastest ← Runner
        ENDIF
    ENDIF
NEXT Runner
```

- **A starting value** like `Fastest ← 0` is a sentinel: the runners are numbered from 1, so 0 can never be a real answer, and if it is still 0 after the loop no runner qualified.
- **The outer IF** picks out the runners that meet the condition (a time under 60 seconds).
- **The inner IF** makes the first qualifying runner the only one recorded: once `Fastest` holds a runner number it is no longer 0, so later runners who also qualify are ignored. Without it, the last qualifier would be kept instead.
- Other questions: which line initialises and which loops, and which CASE clause can never run and why.

13. Choose and justify a loop structure (9 questions)

Name the loop and give a reason **from this scenario**:

- **Count-controlled**: the number of passes is known before the loop starts (the array has 500 elements; the string has `LENGTH(Text)` characters).
- **Pre-condition**: the loop must stop as soon as the item is found or the file ends, and may need to run zero times (an empty file).
- **Post-condition**: the body must run at least once (the value must be input before it can be checked).

"FOR is not the most suitable" usually means: it keeps going after the item has been found, doing comparisons that aren't needed.

14. Write a single statement or condition (7 questions)

Write one complete statement, using the given identifiers and the insert's functions: `JoinDate ← SETDATE(21, 11, 2026)`; `IF DAYINDEX(JoinDate) = 7 THEN` (a Saturday); `Renewal ← SETDATE(DAY(JoinDate), MONTH(JoinDate), YEAR(JoinDate) + 1)` (the same date a year later); `IF YEAR(JoinDate) = YEAR(TODAY()) THEN` (joined this year); `Item.Price ← 4.75`; `CALL ShowBar(20, '*')`. To rebuild a date from the string "DDMMYYYY", take `LEFT(S, 2)`, `MID(S, 3, 2)` and `RIGHT(S, 4)`, convert each with `STR_TO_NUM`, and pass them to `SETDATE`. Don't add the header or `DECLARE` unless asked.

15. Good programming practice (7 questions)

Name a feature and say how it helps: meaningful identifiers (the name says what the data is); comments (explain what each part does); indentation and white space (show where each structure starts and ends); constants; local variables; an array instead of many separate variables (one loop processes them all, and it scales to more items). Each makes the program easier to understand and so easier to debug and maintain. Single letters such as `x` give no meaning.

16. Replace literal values with constants (3 questions)

Declare it once at the start: `CONSTANT MinAge = 16`, then use `MinAge` everywhere. Benefits, said about the scenario: when the value changes it is updated in one place and every use changes; no typing errors in repeated numbers; the name makes the code clearer; the value can't be changed by mistake while the program runs.

Worked examples

The bold codes show where marks are earned, as in Cambridge mark schemes. The number is how many marks.

- **B1** mark for a correct result on its own, with no method needed.

Example 1

The variables in a program hold these values:

Variable	Value
Code	"KT-2048-Q"
Rate	7.25
Joined	14/03/2019 (a DATE)
Active	TRUE

Evaluate each expression. Write ERROR if the expression contains an error. [7]

Expression

LEFT(Code, 2) & RIGHT(Code, 1)

STR_TO_NUM(MID(Code, 4, 4)) DIV 100

INT(Rate * 2) MOD 4

NUM_TO_STR(YEAR(Joined) + 1)

NOT Active OR MONTH(Joined) < 3

LENGTH(Code) + Rate

Code & Rate

Expression	Evaluates to	Working
LEFT(Code, 2) & RIGHT(Code, 1)	"KTQ"	"KT" joined to "Q" B1
STR_TO_NUM(MID(Code, 4, 4)) DIV 100	20	positions 4 to 7 are "2048"; 2048 DIV 100 B1
INT(Rate * 2) MOD 4	2	INT(14.5) is 14; 14 MOD 4 B1
NUM_TO_STR(YEAR(Joined) + 1)	"2020"	a string, so in quotes B1
NOT Active OR MONTH(Joined) < 3	FALSE	FALSE OR (3 < 3, FALSE) B1
LENGTH(Code) + Rate	16.25	9 + 7.25: INTEGER and REAL can be added B1
Code & Rate	ERROR	& can't join a REAL B1

Example 2

Product codes are typed in with spaces, hyphens and lower-case letters, such as "ab-12 cd-34". A function TidyCode () will:

- take the typed code as a STRING parameter;
- remove every space and every hyphen, and change every letter to upper case;
- return the tidied code if it is exactly 8 characters long, otherwise return an empty string.

Write pseudocode for TidyCode (). [7]

```
FUNCTION TidyCode(RawCode : STRING) RETURNS STRING
  DECLARE Index : INTEGER
  DECLARE ThisChar : CHAR
```

```

DECLARE Tidy : STRING
Tidy ← ""
FOR Index ← 1 TO LENGTH(RawCode)
    ThisChar ← MID(RawCode, Index, 1)
    IF ThisChar <> ' ' AND ThisChar <> '-' THEN
        Tidy ← Tidy & TO_UPPER(ThisChar)
    ENDIF
NEXT Index
IF LENGTH(Tidy) <> 8 THEN
    RETURN ""
ENDIF
RETURN Tidy
ENDFUNCTION

```

- **B1** header with a STRING parameter, RETURNS STRING, and ENDFUNCTION;
- **B1** Tidy set to "" before the loop;
- **B1** loop from 1 to the length of the parameter;
- **B1** one character taken with MID in the loop;
- **B1** test that it is neither a space nor a hyphen;
- **B1** join the upper-case character to Tidy;
- **B1** check the length is 8 and return Tidy or "" in both cases.

TidyCode("ab-12 cd-34") returns "AB12CD34", and TidyCode("x-1") returns "".

Example 3

A bike-hire scheme has 250 docks. Each dock is stored in a record:

```

TYPE BikeSlot
    DECLARE BikeID : STRING      // "" if the dock is empty
    DECLARE Charge : INTEGER     // battery charge, 0 to 100
    DECLARE Faulty : BOOLEAN
ENDTYPE
DECLARE Dock : ARRAY[1:250] OF BikeSlot

```

(a) Write efficient pseudocode for a function FindBike() that takes a minimum charge as an INTEGER parameter and returns the index of the first dock holding a bike that is not faulty and has at least that charge, or returns -1 if there is none. [7]

(b) A procedure CountSlots() must give the calling program two values: the number of empty docks and the number of faulty bikes. Write pseudocode for CountSlots(). [5]

(c) Explain why a FOR loop is suitable in (b) but not in (a). [2]

(a)

```

FUNCTION FindBike(MinCharge : INTEGER) RETURNS INTEGER
    DECLARE Index : INTEGER
    DECLARE Found : BOOLEAN
    Index ← 1
    Found ← FALSE
    WHILE Index <= 250 AND NOT Found
        IF Dock[Index].BikeID <> "" AND Dock[Index].Faulty = FALSE AND Dock[Index].Charge >= MinCharge THEN
            Found ← TRUE
        ELSE
            Index ← Index + 1
        ENDIF
    ENDWHILE
    IF Found THEN
        RETURN Index
    ENDIF
ENDFUNCTION

```

```

ELSE
    RETURN -1
ENDIF
ENDFUNCTION

```

- **B1** header with an INTEGER parameter and RETURNS INTEGER;
- **B1** index and flag initialised before the loop;
- **B1** conditional loop through the 250 elements...
- **B1** ...that stops as soon as a bike is found;
- **B1** dot notation with the index;
- **B1** the three tests (dock used, not faulty, enough charge) joined with AND;
- **B1** return the index if found, -1 if not, after the loop.

With bikes only in dock 3 (charge 35, not faulty), dock 7 (charge 90, faulty) and dock 12 (charge 80, not faulty), FindBike(50) returns 12, FindBike(30) returns 3 and FindBike(95) returns -1.

(b) Two values must come back, so this is a procedure with two BYREF parameters:

```

PROCEDURE CountSlots(BYREF EmptyCount : INTEGER, BYREF FaultyCount : INTEGER)
    DECLARE Index : INTEGER
    EmptyCount ← 0
    FaultyCount ← 0
    FOR Index ← 1 TO 250
        IF Dock[Index].BikeID = "" THEN
            EmptyCount ← EmptyCount + 1
        ELSE
            IF Dock[Index].Faulty = TRUE THEN
                FaultyCount ← FaultyCount + 1
            ENDIF
        ENDIF
    NEXT Index
ENDPROCEDURE

```

- **B1** procedure header with two parameters...
- **B1** ...both passed BYREF, so the counts reach the caller;
- **B1** both counts set to 0 before a loop over all 250 docks;
- **B1** count the empty docks;
- **B1** count faulty bikes only in docks that are in use.

With the same three bikes, the caller's variables become 247 empty docks and 1 faulty bike.

(c) In (b) every dock must be checked, so the number of passes (250) is known before the loop starts. **B1** In (a) the search should stop as soon as a suitable bike is found; a FOR loop would go on checking the rest of the docks. **B1**

Example 4

A text file Readings.txt holds one reading per line in the form <SensorID>:<Temperature>, for example "S07:23.5" or "K12:-4.25". Every sensor ID is exactly 3 characters long.

(a) Write pseudocode for a procedure CopyHigh() that takes a REAL parameter Limit, copies every line whose temperature is greater than Limit into a new file Alerts.txt, and then outputs the number of lines copied with a suitable message. [7]

(b) The procedure is to be changed into a function that returns the number of lines copied instead of outputting it. Write the new header and describe the other changes needed, including to the call. [3]

(c) Explain why a pre-condition loop is used in (a) instead of a post-condition loop. [1]

(a)

```

PROCEDURE CopyHigh(Limit : REAL)
  DECLARE Line : STRING
  DECLARE Temp : REAL
  DECLARE Copied : INTEGER
  Copied ← 0
  OPENFILE "Readings.txt" FOR READ
  OPENFILE "Alerts.txt" FOR WRITE
  WHILE NOT EOF("Readings.txt")
    READFILE "Readings.txt", Line
    Temp ← STR_TO_NUM(RIGHT(Line, LENGTH(Line) - 4))
    IF Temp > Limit THEN
      WRITEFILE "Alerts.txt", Line
      Copied ← Copied + 1
    ENDIF
  ENDWHILE
  CLOSEFILE "Readings.txt"
  CLOSEFILE "Alerts.txt"
  OUTPUT Copied, " readings were above ", Limit
ENDPROCEDURE

```

- **B1** Copied initialised before the loop;
- **B1** Readings.txt opened for READ and Alerts.txt for WRITE, both closed after the loop;
- **B1** loop until the end of Readings.txt;
- **B1** a line read inside the loop;
- **B1** the temperature cut out after the 4th character (ID and colon) and converted with STR_TO_NUM;
- **B1** compared with Limit, and the line written and counted if greater;
- **B1** the count output with a message after the loop.

With the lines "S07:23.5", "K12:-4.25", "S03:31.0" and "A90:30.75" and Limit 30, the lines for S03 and A90 are copied and the output is 2 readings were above 30. For "K12:-4.25", RIGHT(Line, 9 - 4) is "-4.25", which STR_TO_NUM turns into -4.25.

(b) FUNCTION CopyHigh(Limit : REAL) RETURNS INTEGER **B1**. Replace the OUTPUT with RETURN Copied and ENDPROCEDURE with ENDFUNCTION. **B1** The call is no longer a CALL statement but part of an expression, such as NumHigh ← CopyHigh(30.0). **B1**

(c) The file might be empty: a pre-condition loop tests EOF before the first read, while a post-condition loop would try to read a line that isn't there. **B1**

Common mistakes

- **No header ending, or the wrong kind of module.** Leaving out ENDFUNCTION or ENDPROCEDURE, writing a function as a procedure, or a procedure where the question says a value is returned. Examiners report the header and ending mark lost every year.
- **OUTPUT instead of RETURN** in a function, or calling a function with CALL, or copying the whole header (with types) into the call. A call gives only the name and the arguments: Total ← AddUp(Prices, 20).
- **Ignoring the parameters.** Using INPUT inside a module that was given its data as a parameter, or opening a file called "FileName" when FileName is a parameter: a parameter name never goes in quotes.
- **Counters not initialised, or initialised inside the loop**, so they restart every pass. Set them once, before the loop. Start a largest value at the first element, not at 0.
- **Malformed loops:** FOR Index 1 TO 10 without ←, a FOR closed with ENDWHILE, a WHILE whose variable never changes (an infinite loop).
- **Using the wrong function:** LEFT to take one character from the middle (use MID); LENGTH on a number or to count lines in a file; RAND(6) alone for a dice roll (it gives a REAL from 0 to just under 6; use INT(RAND(6)) + 1); string functions on a DATE (use DAY, MONTH, YEAR, DAYINDEX); functions from Python.
- **Misusing EOF:** WHILE EOF("Data.txt") = "" or reading the file name. EOF(File) is TRUE or FALSE: loop WHILE NOT EOF(File).

- **An ELSE that undoes a find.** In a search loop, setting `Found ← FALSE` in the ELSE wipes out an earlier match. Only set the flag to TRUE in the loop.
- **Missing quotes when evaluating.** A string result must be in double quotes (" 20 " and 20 are different types).
- **Outputting "not found" inside the loop,** so it appears once for every non-matching element. Decide after the loop.
- **Repeating the question instead of answering it.** "It is more efficient" or "use a loop" earns nothing; say why for this code (it stops when the element is found).
- **Confusing by value and by reference.** If a procedure must hand back a changed value and its parameter isn't BYREF, the caller never sees the change.

How to get the marks

- **"Write pseudocode for ..."** is marked point by point: header and ending with the right parameter and return types; initialisation before the loop; the right loop; each test; the update; the result after the loop. A clear, partly correct answer still collects most of these, so always write the header and the loop.
- **Use the insert.** Learn its functions before the exam; they are the only built-in functions you may use, with the argument types it shows.
- **Use the identifiers in the question** exactly (`Dock`, `BikeID`, `Readings.txt`), and Cambridge syntax: `←`, `ENDIF`, `NEXT Index`, `ENDWHILE`, `UNTIL`.
- **"Efficient"** means a loop that stops as soon as the job is done, no repeated function calls, and no needless tests.
- **"All variables used must be declared"** means a `DECLARE` with a type for every local variable, including the loop index.
- **Evaluate tables:** one mark per row; strings in quotes, Boolean results as TRUE or FALSE, ERROR where a type is wrong.
- **"Explain" or "justify"** needs a reason tied to the code or scenario: "a conditional loop, because the search can stop as soon as the first matching element is found".
- **"State" or "Identify"** wants one short answer: only the first answer on a line is marked.
- **"Describe the steps" or "do not use pseudocode"** wants numbered steps in words, precise enough to code from: name the file and mode, the loop and when it ends, the test, the conversion.
- **Follow-through:** a mistake in one part (such as a wrong identifier) is usually penalised once, so keep going with your own values.

Quick check

- Trace this code and give its output.

```
X ← 2
Y ← 0
WHILE X < 20
  IF X MOD 2 = 1 THEN
    Y ← Y + X
  ENDIF
  X ← X * 2 - 1
ENDWHILE
OUTPUT X, " ", Y
```

- Rewrite this CASE structure as nested IF statements, combining the first two clauses.

```
CASE OF Zone
  'N' : Fee ← 4.50
  'S' : Fee ← 4.50
  'E' : Fee ← 6.00
  OTHERWISE : Fee ← 0
ENDCASE
```

- Write the headers for: (a) a function `PriceAfter ( )` that takes a price and a Boolean saying whether the customer is a member, and returns the price to pay; (b) a procedure `Swap ( )` that exchanges the values of two of the caller's integer variables. Explain your choice of parameter passing in (b).

4. Each statement contains one error. Identify it and write the correct statement. (i) `IF Count ← 10 THEN` (ii) `Size ← LENGTH(472)` (iii) `Label ← "Total: " & 85`
5. A program asks a user to enter their 4-digit PIN, allowing at most 3 attempts. Name the most suitable loop, justify your choice, and write its ending condition using `Entered`, `Stored` and `Attempts`.

Answers

1. X goes 2, 3, 5, 9, 17, 33. Y adds each odd X before it changes: $3 + 5 + 9 + 17 = 34$. When X is 33 the loop ends. The output is 33 34.
2.

```
IF Zone = 'N' OR Zone = 'S' THEN
    Fee ← 4.50
ELSE
    IF Zone = 'E' THEN
        Fee ← 6.00
    ELSE
        Fee ← 0
    ENDIF
ENDIF
```
3. (a) `FUNCTION PriceAfter(Price : REAL, IsMember : BOOLEAN) RETURNS REAL.` (b) `PROCEDURE Swap(BYREF First : INTEGER, BYREF Second : INTEGER).` The changes must be made to the caller's own variables, so both are passed by reference; by value, only copies would be swapped and the caller's variables would not change.
4. (i) ← assigns; a condition compares with =: `IF Count = 10 THEN.` (ii) `LENGTH` needs a string: `Size ← LENGTH(NUM_TO_STR(472))`, which gives 3. (iii) & joins strings only: `Label ← "Total: " & NUM_TO_STR(85).`
5. A post-condition loop (`REPEAT ... UNTIL`), because the PIN must be entered at least once before it can be checked. `UNTIL Entered = Stored OR Attempts = 3.`

Past questions to try

- [9618/22/M/J/24 Q5](#): spotting and correcting errors in pseudocode statements.
- [9618/21/O/N/23 Q1](#): a CASE structure: dry run it and find the clause that can never run.
- [9618/22/M/J/23 Q2](#): single statements with the date functions.
- [9618/22/M/J/25 Q7](#): an efficient search of a sorted 2D array, and an average.