0 1 An algorithm, that uses the modulus operator, has been represented using pseudo-code in **Figure 1**.

- Line numbers are included but are not part of the algorithm.

Figure 1

```
1  i ← USERINPUT
2  IF i MOD 2 = 0 THEN
3     OUTPUT i * i
4  ELSE
5     OUTPUT i
6  ENDIF
```

The modulus operator is used to calculate the remainder after dividing one integer by another.

For example:

- `14 MOD 3` evaluates to `2`
- `24 MOD 5` evaluates to `4`

0 1 . 1 Shade **one** lozenge that shows the line number where selection is **first** used in the algorithm in **Figure 1**.

[1 mark]

A Line number 1 ⬭

B Line number 2 ⬭

C Line number 3 ⬭

D Line number 4 ⬭

0 1 . 2 Shade **one** lozenge that shows the output from the algorithm in **Figure 1** when the user input is `4`

[1 mark]

A `0`

B `2`

C `4`

D `8`

E `16`

0 1 . 3 Shade **one** lozenge that shows the line number where assignment is **first** used in the algorithm in **Figure 1**.

[1 mark]

A Line number 1

B Line number 2

C Line number 3

D Line number 4

0 1 . 4 Shade **one** lozenge that shows the line number that contains a relational operator in the algorithm in **Figure 1**.

[1 mark]

A Line number 1

B Line number 2

C Line number 3

D Line number 4

Figure 1 has been included again below.

Figure 1

```
i ← USERINPUT
IF i MOD 2 = 0 THEN
   OUTPUT i * i
ELSE
   OUTPUT i
ENDIF
```

0 1 . 5 Shade **one** lozenge to show which of the following is a **true** statement about the algorithm in **Figure 1**.

[1 mark]

A This algorithm uses a Boolean operator.

B This algorithm uses a named constant.

C This algorithm uses iteration.

D This algorithm uses the multiplication operator.

0 1 . 6 **Figure 2** shows an implementation of the algorithm in **Figure 1** using the C# programming language.

- Line numbers are included but are not part of the program.

Figure 2

```
Console.Write("Enter a number: ");
int i = Convert.ToInt32(Console.ReadLine());
if (i % 2 == 0) {
   Console.WriteLine(i * i);
}
else {
   Console.WriteLine(i);
}
```

The program in **Figure 2** needs to be changed so that it repeats five times using **definite** (count controlled) iteration.

Shade **one** lozenge next to the program that does this correctly.

[1 mark]

A	<pre>for (int x = 0; x < 5; x++) {
	   Console.Write("Enter a number: ");
	   int i = Convert.ToInt32(Console.ReadLine());
	   if (i % 2 == 0) {
	       Console.WriteLine(i * i);
	   }
	   else {
	       Console.WriteLine(i);
	   }
	}</pre>
		⬭

B	<pre>for (int x = 0; x < 6; x++) {
	   Console.Write("Enter a number: ");
	   int i = Convert.ToInt32(Console.ReadLine());
	   if (i % 2 == 0) {
	       Console.WriteLine(i * i);
	   }
	   else {
	      Console.WriteLine(i);
	   }
	}</pre>
		⬭

C	<pre>int x = 1;
	while (x != 6) {
	   Console.Write("Enter a number: ");
	   int i = Convert.ToInt32(Console.ReadLine());
	   if (i % 2 == 0) {
	       Console.WriteLine(i * i);
	   }
	   else {
	       Console.WriteLine(i);
	   }
	   x = x + 1;
	}</pre>
		⬭

D	<pre>int x = 6;
	while (x != 0) {
	   Console.Write("Enter a number: ");
	   int i = Convert.ToInt32(Console.ReadLine());
	   if (i % 2 == 0) {
	      Console.WriteLine(i * i);
	   }
	   else {
	       Console.WriteLine(i);
	   }
	   x = x - 1;
	}</pre>
		⬭

0 2 **Figure 3** shows an algorithm, represented using pseudo-code, that calculates the delivery cost for an order from a takeaway company.

Figure 3

```
orderTotal ← USERINPUT
deliveryDistance ← USERINPUT
deliveryCost ← 0.0
messageOne ← "Minimum spend not met"
messageTwo ← "Delivery not possible"
IF deliveryDistance ≤ 5 AND orderTotal > 0.0 THEN
   IF orderTotal > 50.0 THEN
      deliveryCost ← 1.5
      OUTPUT deliveryCost
   ELSE IF orderTotal > 25.0 THEN
      deliveryCost ← (orderTotal / 10) * 2
      OUTPUT deliveryCost
   ELSE
      OUTPUT messageOne
   ENDIF
ELSE
   OUTPUT messageTwo
ENDIF
```

0 2 . 1 Using **Figure 3**, complete the table.

[2 marks]

Input value of `orderTotal`	**Input value of** `deliveryDistance`	**Output**
55.5	2	
35.0	5	

0 2 . 2 State how many possible values the result of the comparison `deliveryDistance ≤ 5` could have in the algorithm shown in **Figure 3**.

[1 mark]

____

0 2 . 3 State the most suitable data type for the following variables used in **Figure 3**.

[2 marks]

Variable identifier	Data type
`deliveryCost`	
`messageOne`	

0 2 . 4 State **one** other common data type that you have **not** given in your answer to Question **02.3**.

[1 mark]

Turn over for the next question

0 3 A programmer has written a C# program that asks the user to input two integers and then output which of the two integers is the largest.
Complete the program by filling in the gaps using the information in **Figure 3**. Each item in **Figure 3** should only be used once.

[5 marks]

Figure 3

`Console.Write`	`num1`	`num2`	`output`
`else`	`<`	`>`	`else if`
`string`	`double`	`int`	

```
int num1;

__________ num2;

Console.WriteLine("Enter a number: ");

num1 = int.Parse(Console.ReadLine());

Console.WriteLine("Enter another number: ");

num2 = int.Parse(Console.ReadLine());

if (num1 > num2)

{

    Console.WriteLine("   __________ is bigger.");

}

else

if (num1 ______________ num2)

{

    Console.WriteLine("   __________ is bigger.");

}

____________________

{

    Console.WriteLine("The numbers are equal.");

}
```

0 4 The algorithm in **Figure 1** has been developed to automate the quantity of dog biscuits to put in a dog bowl at certain times of the day. The algorithm contains an error.

- Line numbers are included but are not part of the algorithm.

Figure 1

```
1     time ← USERINPUT
2     IF time = 'breakfast' THEN
3        q ← 1
4     ELSE IF time = 'lunch' THEN
5        q ← 4
6     ELSE IF time = 'dinner' THEN
7        a ← 2
8     ELSE
9        OUTPUT 'time not recognised'
10    ENDIF
11    FOR n ← 1 TO q
12       IF n < 3 THEN
13          DISPENSE_BISCUIT('chewies')
14       ELSE
15          DISPENSE_BISCUIT('crunchy')
16       ENDIF
17    ENDFOR
```

0 4 . 1 Shade **one** lozenge which shows the line number where selection is **first** used in the algorithm shown in **Figure 1**.

[1 mark]

A Line number 2

B Line number 4

C Line number 9

D Line number 12

0 4 . 2 Shade **one** lozenge which shows the line number where iteration is **first** used in the algorithm shown in **Figure 1**.

[1 mark]

A Line number 1

B Line number 8

C Line number 11

D Line number 13

0 4 . 3 Shade **one** lozenge which shows how many times the subroutine `DISPENSE_BISCUIT` would be called if the user input is `'breakfast'`.

[1 mark]

A 1 subroutine call

B 2 subroutine calls

C 3 subroutine calls

D 4 subroutine calls

0 4 . 4 Shade **one** lozenge which shows the data type of the variable `time` in the algorithm shown in **Figure 1**.

[1 mark]

A Date/Time

B String

C Integer

D Real

0 4 . 5 State how many times the subroutine `DISPENSE_BISCUIT` will be called with the parameter `'chewies'` if the user input is `'lunch'`.

[1 mark]

---

0 4 . 6 State how many possible values the result of the comparison `time = 'dinner'` could have in the algorithm shown in **Figure 1**.

[1 mark]

---

0 4 . 7 The programmer realises they have made a mistake. State the line number of the algorithm shown in **Figure 1** where the error has been made.

[1 mark]

---

0 4 . 8 Write **one** line of code that would correct the error found in the algorithm in **Figure 1**.

[1 mark]

---

0 5 Run length encoding (RLE) is a form of compression that creates frequency/data pairs to describe the original data.

For example, an RLE of the bit pattern `00000011101111` could be `6 0 3 1 1 0 4 1` because there are six `0`s followed by three `1`s followed by one `0` and finally four `1`s.

The algorithm in **Figure 7** is designed to output an RLE for a bit pattern that has been entered by the user.

Five parts of the code labelled **L1**, **L2**, **L3**, **L4** and **L5** are missing.

- Note that indexing starts at zero.

Figure 7

```
pattern ← L1
i ← L2
count ← 1
WHILE i < LEN(pattern)-1
   IF pattern[i] L3 pattern[i+1] THEN
      count ← count + 1
   ELSE
      L4
      OUTPUT pattern[i]
      count ← 1
   ENDIF
   L5
ENDWHILE
OUTPUT count
OUTPUT pattern[i]
```

0 5 . 1 Shade **one** lozenge to show what code should be written at point **L1** of the algorithm.

[1 mark]

A `OUTPUT` ⬭

B `'RLE'` ⬭

C `True` ⬭

D `USERINPUT` ⬭

0 5 . 2 Shade **one** lozenge to show what value should be written at point `L2` of the algorithm.

[1 mark]

A `-1` ⬭

B `0` ⬭

C `1` ⬭

D `2` ⬭

0 5 . 3 Shade **one** lozenge to show what operator should be written at point `L3` of the algorithm.

[1 mark]

A = ⬭

B ≤ ⬭

C < ⬭

D ≠ ⬭

0 5 . 4 Shade **one** lozenge to show what code should be written at point `L4` of the algorithm.

[1 mark]

A `count` ⬭

B `count ← count - 1` ⬭

C `count ← USERINPUT` ⬭

D `OUTPUT count` ⬭

0 5 . 5 Shade **one** lozenge to show what code should be written at point **L5** of the algorithm.

[1 mark]

A `i ← i * 2` ⬭

B `i ← i + 1` ⬭

C `i ← i + 2` ⬭

D `i ← i DIV 2` ⬭

0 5 . 6 State a run length encoding of the series of characters `ttjjeeess`

[2 marks]

____

____

____

____

0 5 . 7 A developer implements the algorithm shown in **Figure 7** and tests their code to check that it is working correctly. The developer tests it only with the input bit pattern that consists of six zeros and it correctly outputs `6 0`.

Using example test data, state **three** further tests that the developer could use to improve the testing of their code.

[3 marks]

____

____

____

____

____

____

____

0 6 **Figure 2** shows an algorithm, represented using pseudo-code.

- Line numbers are included but are not part of the algorithm.

Figure 2

```
1    num ← USERINPUT
2    IF NOT(num > 1) OR num > 20 THEN
3       OUTPUT "False"
4    ELSEIF num > 1 AND num < 15 THEN
5       OUTPUT "Almost"
6    ELSEIF num MOD 5 = 0 THEN
7       OUTPUT "True"
8    ELSE
9       OUTPUT "Unknown"
10   ENDIF
```

The modulus operator is used to calculate the remainder after dividing one integer by another.

For example:
- `14 MOD 3` evaluates to `2`
- `24 MOD 5` evaluates to `4`

0 6 . 1 Where is a relational operator **first** used in the algorithm in **Figure 2**?

Shade **one** lozenge.

[1 mark]

A Line number 1 ⬭

B Line number 2 ⬭

C Line number 3 ⬭

D Line number 6 ⬭

0 7 A program is being written to solve a sliding puzzle.

- The sliding puzzle uses a 3 x 3 board.
- The board contains eight tiles and one blank space.
- Each tile is numbered from 1 to 8
- On each turn, a tile can only move one position up, down, left, or right.
- A tile can only be moved into the blank space if it is next to the blank space.
- The puzzle is solved when the tiles are in the correct final positions.

Figure 10 shows an example of how the tiles might be arranged on the board at the start of the game with the blank space in the position (0, 1).

Figure 11 shows the correct final positions for the tiles when the puzzle is solved.

The blank space (shown in black) is represented in the program as number 0

Figure 10

row \ column	0	1	2
0	4	(blank)	2
1	1	7	6
2	5	3	8

Figure 11

row \ column	0	1	2
0	1	2	3
1	4	5	6
2	7	8	(blank)

Table 3 describes the purpose of three subroutines the program uses.

Table 3

Subroutine	**Purpose**
`getTile(row, column)`	Returns the number of the tile on the board in the position `(row, column)` For example: • `getTile(1, 0)` will return the value `5` if it is used on the board in **Figure 12** • `getTile(1, 2)` will return the value `0` if it is used on the board in **Figure 12**.
`move(row, column)`	Moves the tile in position `(row, column)` to the blank space, if the blank space is next to that tile. If the position `(row, column)` is not next to the blank space, no move will be made. For example: • `move(0, 2)` would change the board shown in **Figure 12** to the board shown in **Figure 13** • `move(2, 0)` would not make a move if used on the board shown in **Figure 12**.
`displayBoard()`	Displays the board showing the current position of each tile.

Figure 12

row \ column	0	1	2
0	1	7	4
1	5	8	
2	6	2	3

Figure 13

row \ column	0	1	2
0	1	7	
1	5	8	4
2	6	2	3

0 7 . 1 The C# program shown in **Figure 14** uses the subroutines in **Table 3**, on page 25.

The program is used with the board shown in **Figure 15**.

Figure 14

```
if (getTile(1, 0) == 0)
{
   move(2, 0);
}
if (getTile(2, 0) == 0)
{
   move(2, 1);
}
displayBoard();
```

Figure 15

column

		0	1	2
	0	1	8	3
row	1		7	5
	2	4	2	6

Complete the board to show the new positions of the tiles after the program in **Figure 14** is run.

[2 marks]

column

		0	1	2
	0			
row	1			
	2			