

Paper 1 – Worked Solutions & Mark Scheme

AQA A-level Computer Science 7517/1 · Wednesday 10 June 2026

Total: 100 marks. Each question shows the worked solution followed by a mark scheme (green box). **M** = method, **A** = accuracy, **B** = independent point; / separates acceptable alternatives.

These are independently prepared model answers and indicative mark allocations, not the official AQA mark scheme.

Important - Skeleton Program / Preliminary Material. Sections C and D depend on the Ant simulation Skeleton Program and Preliminary Material, which were not included in the files provided. Answers in those sections are therefore **provisional**: Section C answers are inferred from the wording of the questions, and Section D solutions are written as language-neutral Python-style code using placeholder names. They are flagged with an amber box and must be checked against the actual skeleton code (identifiers, language, grid indexing).

Contents: Section A (Q1–Q4, 37 marks) · Section B (Q5, 13 marks) · Section C (Q6–Q10, 13 marks, provisional) · Section D (Q11–Q14, 37 marks, provisional code)

Section A

Question 1 - Turing machine (10 marks)

01.1 [1 mark]

What is the maximum size for the tape of a Turing machine?

Worked solution

The tape is unbounded in both directions – it is of **infinite** length.

Mark scheme

B1 Infinite / unlimited / unbounded

01.2 [6 marks]

Trace the Turing machine for the first 23 steps.

Worked solution

Apply the transition function at each step: write the new symbol, move the head, change state. (× is a marker for a 1 that has been counted; – is blank.)

Step								State
0	–	–	–	1*	1	1	–	S0
1	–	–	–*	×	1	1	–	S1
2	–	–	1	×	1	1	–	S0
3	–	–	1	×	1*	1	–	S0
4	–	–	1	×	×	1	–	S1
5	–	–	1*	×	×	1	–	S1
6	–	–*	0	×	×	1	–	S1
7	–	1	0*	×	×	1	–	S0
8	–	1	0	×	×	1	–	S0
9	–	1	0	×	×	1	–	S0
10	–	1	0	×	×	1*	–	S0
11	–	1	0	×	×	×	–	S1
12	–	1	0	×	×	×	–	S1
13	–	1	0*	×	×	×	–	S1
14	–	1	1	×	×	×	–	S0
15	–	1	1	×	×	×	–	S0
16	–	1	1	×	×	×	–	S0
17	–	1	1	×	×	×	–*	S0
18	–	1	1	×	×	×	–	S2
19	–	1	1	×	×	–	–	S2
20	–	1	1	×	–	–	–	S2
21	–	1	1*	–	–	–	–	S2
22	–	1	1	–*	–	–	–	S3
23	–	1	1*	–	–	–	–	S4

Mark scheme

Award marks for correct rows in the table (6 marks): **B1** rows 1–3 correct (first moves S0→S1→S0) · **B1** rows 4–7 correct · **B1** rows 8–14 correct (× markers, 0 → 1 change) · **B1** rows 15–18 correct (head moves right, S0→S2) · **B1** rows 19–22 correct (erasing ×, S2→S3) · **B1** row 23 correct – tape – 1 1 – – –, head on the second cell, state S4 (halt). *Error carried forward: a single early error is penalised once; later rows consistent with it can still earn marks.*

01.3 [1 mark]

State the purpose of the Turing machine in Figure 1.

Worked solution

The final tape is 11, which is binary 3 – the starting tape held three 1s (unary 3). Testing input 1 → 1 and input 11 → 10 confirms it: the machine converts a **unary number to binary** (it counts the 1s and writes a binary counter).

Mark scheme

B1 Converts a unary number to binary / counts the number of 1s and outputs the count in binary

01.4 [2 marks]

Describe the importance of Turing machines to computation theory.

Worked solution

A Turing machine is a simple mathematical model of computation. It defines what is computable: any problem that can be solved by an algorithm can be solved by a Turing machine (Church–Turing thesis), so it is used to reason about the limits of computation, e.g. to prove a problem such as the halting problem is non-computable. A universal Turing machine can simulate any other, which underpins the idea of a general-purpose, stored-program computer.

Mark scheme

B1 It is a model / definition of computation – a problem is computable if (and only if) a Turing machine can solve it

B1 Used to show that some problems cannot be computed (e.g. halting problem) / a universal Turing machine can simulate any other (basis of the modern computer)

Question 2 – Queues and data structures (11 marks)

02.1 [5 marks]

Describe the steps to add an item to a circular queue (including how pointers are used or changed).

Worked solution

The queue is a fixed-length array with a front pointer, a rear pointer and (usually) a size/count of items.

1. Check whether the queue is full (size = array length). If it is, report an error / do not add the item. 2. Otherwise move the rear pointer on by one: $\text{rear} \leftarrow (\text{rear} + 1) \text{ MOD length}$ – so it wraps back to position 0 after the end of the array. 3. Store the new item in the array at the position given by the rear pointer. 4. Increase the size (count of items) by 1. The front pointer does not change on an enqueue.

Mark scheme

B1 Check whether the queue is full (e.g. size = maximum / $\text{rear} + 1 \text{ MOD length} = \text{front}$) and if so do not add / report error

B1 Rear pointer is incremented (moved to the next free position)

B1 If the rear pointer goes past the end of the array it wraps round to the start (0) / uses MOD

B1 Item is stored in the array at the position of the rear pointer

B1 Size / count of the queue is incremented by 1 (front pointer unchanged)

02.2 [3 marks]

Describe the steps to add an item to a priority queue (no need to describe pointers).

Worked solution

Check the queue is not full. Work along the queue to find the correct position for the new item according to its priority (after all items of higher or equal priority). Shift the items from that position onwards (lower priority) one place along to make space, then insert the new item in the gap.

Mark scheme

B1 Check the queue is not full

B1 Find the correct position for the item based on its priority (comparing with existing items; after items of equal priority)

B1 Move lower-priority items along one place and insert the item in the correct position (alt: add at end and re-sort by priority)

02.3 [3 marks]

Describe three differences between dynamic and static data structures.

Static	Dynamic
Fixed size, set at compile time / when declared	Size can grow and shrink at run time
Memory allocated at compile time; may waste memory or run out of space	Memory allocated as needed (from the heap) so uses only what is required
Elements stored in contiguous memory (fast direct access by index)	Elements may be scattered in memory and linked by pointers
Simpler to implement; no memory management needed	More complex; memory must be freed/garbage-collected, overhead of pointers

Mark scheme

B1 x 3 Any three valid differences (each must contrast both structures): fixed vs variable size · compile-time vs run-time allocation · contiguous vs non-contiguous storage/pointers · wasted/unused memory vs efficient use · stack vs heap

Question 3 – Complexity and trees (12 marks)

03.1 [1 mark]

What is wrong with “Big-O is used to express how the time taken by an algorithm grows relative to the amount of memory it uses”?

Worked solution

Big-O expresses how the time taken (or space used) grows relative to the **size of the input** (n), not the amount of memory used.

Mark scheme

B1 Big-O compares time with the size of the input (number of items), not the memory used

03.2 [1 mark]

Big-O of f (recursive factorial).

Worked solution

$f(n)$ makes one recursive call for each value $n, n-1, \dots, 1$, so the number of calls grows linearly: **$O(n)$** .

Mark scheme

B1 $O(n)$

03.3 [1 mark]

Big-O of g .

Worked solution

g performs three assignments whatever n is, so constant time: **$O(1)$** .

Mark scheme

B1 $O(1)$

03.4 [1 mark]

Big-O of h .

Worked solution

Nested FOR loops each run n times, so the inner statement executes $n \times n$ times: **$O(n^2)$** .

Mark scheme

B1 $O(n^2)$ (accept $O(n^2)$)

03.5 [1 mark]

Maximum comparisons for a binary tree search on a full binary tree of 31 nodes.

Worked solution

A full tree with 31 nodes has 5 levels ($1 + 2 + 4 + 8 + 16 = 31$). In the worst case the search reaches a leaf: **5** comparisons.

Mark scheme

B1 5

03.6 [1 mark]

Maximum comparisons on any binary tree of 31 nodes.

Worked solution

The worst-case tree is a degenerate (linear) tree – a chain – with 31 levels, so up to **31** comparisons.

Mark scheme

B1 31

03.7 – 03.9 [6 marks]

Traversals of the tree in Figure 3 (F root; K and L children; R, B under K; P, M under L).

Traversal	Rule	Order visited
In-order (03.7)	left, root, right	R K B F P L M
Pre-order (03.8)	root, left, right	F K R B L P M
Post-order (03.9)	left, right, root	R B K P M L F

Mark scheme

Each traversal – **2 marks** for the fully correct order; **1 mark** if the order is correct except for at most one pair of nodes swapped / the correct start and a sensible traversal attempted

Question 4 – Regular expressions (4 marks)

04.1 [1 mark]

A string matching both $1+0|10?11$ and $101^*(01)?$.

Worked solution

Expression 1 is $(1+0)$ or $(10?11)$: strings such as 10, 110, 1110 (one or more 1s then 0) or 111, 1011.

Expression 2 is 1, 0, then any number of 1s, then optionally 01: e.g. 10, 101, 1011, 1001. Strings in both: **10** and **1011**.

Mark scheme

B1 10 or 1011 (or any string matching both)

04.2 [3 marks]

Which expressions match 111 as well as everything $101^*(01)?$ matches?

Worked solution

For each, test (a) does it match 111 and (b) does it match all strings of the form $101^*(01)?$

Expression	Matches 111?	Matches all of $101^*(01)?$	Y/N
$101^*(01)? 111$	yes (second alternative)	yes (first alternative)	Y
$10?1^*(01)?$	1, no 0, $1^*=11$	$0?$ matches the 0	Y
$101^*(01)?1+$	no – must start 10	–	N
$10^*1^*(01)?$	1, 0^* empty, $1^*=11$	0^* matches the single 0	Y
$101(11)^*(01)?$	no – must start 10	–	N
$(111)(10?)101^*(01)?$	no – 111 then 1 then 10...	no – requires the prefix 111	N

Mark scheme

B3 All six correct: Y, Y, N, Y, N, N **B2** four or five correct **B1** two or three correct

Section B

Question 5 – Program: squares formed by number + digit sum (13 marks)

05.1 [12 marks]

Write a program that repeats input until $n > 1$, then outputs every number from 2 to n for which the number plus the sum of its digits is a square number.

Worked solution

Solution (Python): validation loop, a digit-sum calculation, a perfect-square test using integer square root, and a loop from 2 to n inclusive (the example includes 17).

```
import math

def is_square(x):
    r = math.isqrt(x)          # integer square root
    return r * r == x

def digit_sum(x):
    total = 0
    while x > 0:
        total = total + x % 10 # last digit
        x = x // 10           # remove last digit
    return total

n = int(input("Enter a limiting value: "))
while n <= 1:                  # keep asking until greater than 1
    print("The value must be greater than 1.")
    n = int(input("Enter a limiting value: "))

for number in range(2, n + 1): # 2 to n inclusive
    if is_square(number + digit_sum(number)):
        print(number)
```

Mark scheme

B1 Input of n as an integer **B1** Iteration (loop) to repeat input **B1** Condition correct: continues while $n \leq 1$ (stops when $n > 1$)
B1 Loop through numbers 2 ... n *inclusive*
B1 Extracts digits correctly (e.g. MOD 10 and DIV 10, or via string)
B1 Digit sum calculated **B1** Adds the digit sum to the number
B1 Test for a square number (e.g. integer root squared = value / sqrt is whole number)
B1 Outputs the number (not the sum) only when the test is true **B1** Correct output for the test data **B1** Sensible use of subroutines/variables; no errors **B1** Code compiles/runs and is logically structured

05.2 [1 mark]

Screen captures for the inputs 1 then 245.

Worked solution

The program rejects 1 and asks again, then prints these numbers for n = 245 (checked by running the code above):

```
Enter a limiting value: 1
The value must be greater than 1.
Enter a limiting value: 245
2
8
17
27
38
72
86
135
161
179
216
245
```

Mark scheme

B1 Screen capture showing 1 rejected (re-prompt) and then 245 giving exactly: 2, 8, 17, 27, 38, 72, 86, 135, 161, 179, 216, 245

Section C (provisional - needs the Skeleton Program)

Provisional answers. Questions 6–10 refer to the Preliminary Material and Skeleton Program, which were not supplied. Answers marked ✓ are general theory (safe). Answers marked △ are inferred from the wording of the question and must be confirmed against the code. If you send me the Skeleton Program (and Preliminary Material) I can finalise them.

06 [3 marks]

Three circumstances (other than picking up/adding food) when an ant will not move to the neighbouring cell with the strongest own-nest pheromone. △

Worked solution

Likely valid circumstances (verify against ChooseCellToMoveTo / ChangeCell): (1) none of the neighbouring cells contains any pheromone belonging to the ant's own nest, so there is no 'strongest' cell and the ant moves randomly; (2) the cell with the strongest pheromone is not a valid neighbour – it would be off the edge of the grid (index -1) or is the ant's current cell; (3) the ant is of a type that does not follow pheromone (e.g. a queen/other ant type with its own movement rule) or the ant has a random chance of ignoring the trail; (4) the ant has already been removed/culled or can't move this stage.

Mark scheme

B1 × 3 Any three correct circumstances from the skeleton's logic (above are indicative; mark scheme will follow the code)

07.1 [1 mark]

What does adding -1 to ListOfNeighbours indicate? △

Worked solution

That there is no valid neighbouring cell at that position (the position lies outside the grid).

Mark scheme

B1 There is no neighbour in that direction / the neighbouring cell would be outside the grid (invalid cell)

07.2 [1 mark]

Why will the 5th element always be -1? △

Worked solution

The neighbours are the cells in the 3 × 3 block around the ant; the 5th (middle) element would be the ant's own cell, which is not a neighbour, so it is always set to -1.

Mark scheme

B1 The 5th element represents the ant's own / current cell (centre of the 3×3 block), which is not a neighbour

08.1 [1 mark]

What is overriding? ✓

Worked solution

A subclass provides its own version of a method that is already defined in its parent (super) class, with the same name and parameters, replacing the inherited behaviour.

Mark scheme

B1 A method in a subclass with the same name/signature as in the superclass that replaces/redefines the inherited method

08.2 [1 mark]

Two inherited properties/attributes used in AdvanceStage. △

Worked solution

Answer from the Nest class: attributes it declares itself are *not* inherited. Choose two attributes used in AdvanceStage that are declared in the parent class (e.g. the location/ID attributes) – need the code to name them.

Mark scheme

B1 Two attribute names taken from the parent class and used in the AdvanceStage subroutine (names from skeleton)

08.3 [1 mark]

Difference between private and protected specifiers. ✓

Worked solution

A private member can be accessed only within the class that declares it; a protected member can be accessed within that class *and its subclasses* (derived classes).

Mark scheme

B1 Private – only accessible in the declaring class; protected – accessible in the class and in classes that inherit from it

08.4 [1 mark]

Final value of AntsToCull with no food and 10 ants. ⚠

Worked solution

Trace the AntsToCull calculation in AdvanceStage with Food = 0 and 10 ants – formula not visible without the code.

Mark scheme

B1 Value from tracing the code (needs skeleton)

08.5 [1 mark]

How many of the five statements about AdvanceStage are true? ⚠

Worked solution

Each statement has to be checked against the AdvanceStage code (probability of creating queens vs workers, and the number created per call).

Mark scheme

B1 Correct count (needs skeleton)

09 [2 marks]

Two differences between global and local variables. ✓

Worked solution

A global variable is declared outside any subroutine and can be accessed from anywhere in the program, existing for the whole run. A local variable is declared inside a subroutine, is accessible only there, and exists only while that subroutine is executing (memory then released). Using local variables avoids unintended side effects and name clashes.

Mark scheme

B1 Scope – global accessible throughout the program; local only within the subroutine/block where declared

B1 Lifetime – global exists for the whole program run; local created when the subroutine is called and destroyed when it ends

10 [1 mark]

Difference between composition and aggregation. ✓

Worked solution

In composition the part cannot exist without the whole (the whole owns it and it is destroyed with it); in aggregation the part can exist independently of the whole.

Mark scheme

B1 Composition – part's lifetime depends on / is owned by the whole; aggregation – part can exist independently of the whole

Section D (indicative code – adapt to the Skeleton Program)

Provisional code. The Skeleton Program (language, class and attribute names, grid indexing) was not provided, so the solutions below are written in Python-style code with *placeholder names* (e.g. Cell, self.Grid, GetFood()) that need to be matched to the real skeleton. The logic and the mark points are what matters; they are structured to match the tasks printed in the paper.

11 [5 marks]

Add food to any cell (DisplayMenu and Main).

Worked solution

Task 1: display the new option. Task 2: in Main, if chosen, input the row and column and add 500 food to that cell.

```
def DisplayMenu():  
    print("1. Run to end of simulation")  
    ... # existing options
```

```

        print("N. Add food to a cell")          # NEW (N = next free option number)

# in Main, where the menu choice is processed
elif Choice == NEW_OPTION:
    Row = int(input("Enter row: "))
    Column = int(input("Enter column: "))
    TheSimulation.AddFood(Row, Column, 500)    # or: cell.UpdateFoodInCell(500) / existing food + 500

```

Mark scheme

- B1** New option displayed in DisplayMenu
- B1** Main handles the new option (selection/branch added)
- B1** Row and column input
- B1** 500 added to (not replacing) the food already in that cell

Test (11.2, 1 mark): screen capture showing the menu with the new option, food added to an empty cell (500), added again (1000) and after entering 3 (food reduced/changed accordingly)

12 [8 marks]

GetAreaDetails as a grid.

Worked solution

Each cell is printed in a fixed 4-character field so every cell takes the same space: character 1 = N (nest), 2 = A/B/C for 1/2/more than 2 ants, 3 = P (pheromone), 4 = F (food); empty slots are spaces. A '|' is printed before each cell and at the end of each row, and each row is on a new line.

```

def GetAreaDetails(self, StartRow, StartColumn, EndRow, EndColumn):
    Details = ""
    for Row in range(StartRow, EndRow + 1):
        for Column in range(StartColumn, EndColumn + 1):
            Cell = self.GetCell(Row, Column)          # placeholder accessors
            Text = ""
            Text += "N" if Cell.ContainsNest() else " "
            Ants = Cell.NumberOfAnts()
            if Ants == 0:
                Text += " "
            elif Ants == 1:
                Text += "A"
            elif Ants == 2:
                Text += "B"
            else:
                Text += "C"
            Text += "P" if Cell.PheromoneCount() > 0 else " "
            Text += "F" if Cell.FoodAmount() > 0 else " "
            Details += "|" + Text                      # '|' before every cell
        Details += "|" + "\n"                          # '|' at end of row, then new line
    return Details

```

Mark scheme

- B1** Nested loops over rows and columns of the selected area (inclusive of end row/column)
- B1** N shown if there is a nest **B1** A / B / C chosen correctly by number of ants (1, 2, more than 2)
- B1** P if at least one pheromone **B1** F if there is food
- B1** Fixed width per cell (padded with spaces) so the grid lines up
- B1** '|' before each cell and at end of each row; each row on a new line (*old list code removed*)

Test (12.2, 1 mark): screen capture of the 5 × 5 grid for rows 1-5, columns 1-5

13 [12 marks]

Obstacles (Cell, Simulation.GetIndicesOfNeighbours, AddObstaclesToColumn, Main).

```

class Cell:
    def __init__(self, ...):
        ...
        self.__HasObstacle = False                # private, False in constructor

    def ContainsAnObstacle(self):
        return self.__HasObstacle

    def AddObstacle(self):
        self.__HasObstacle = True

class Simulation:

```

```

def AddObstaclesToColumn(self, Column):
    for Row in range(1, self.__NoOfRows + 1):      # every row in the column
        self.__Grid[Row][Column].AddObstacle()

def GetIndicesOfNeighbours(self, Row, Column):
    ListOfNeighbours = []
    for RowDiff in (-1, 0, 1):
        for ColDiff in (-1, 0, 1):
            R, C = Row + RowDiff, Column + ColDiff
            if (RowDiff, ColDiff) != (0, 0) and cell_in_grid(R, C) \
                and not self.__Grid[R][C].ContainsAnObstacle():    # NEW test
                ListOfNeighbours.append((R, C))
            else:
                ListOfNeighbours.append(-1)        # keep the existing -1 convention
    return ListOfNeighbours

# Main, where simulation number 2 is set up
if SimulationNumber == 2:
    ...
    TheSimulation.AddObstaclesToColumn(3)    # NEW

```

Mark scheme

B1 Cell: private HasObstacle attribute **B1** Set to False in the constructor

B1 ContainsAnObstacle returns HasObstacle **B1** AddObstacle sets it to True

B1 AddObstaclesToColumn defined with one integer parameter **B1** Loops over all rows of that column **B1** Calls AddObstacle on each cell

B1 GetIndicesOfNeighbours checks ContainsAnObstacle for the neighbouring cell **B1** Cell with an obstacle excluded (−1 / not added) **B1** Existing behaviour otherwise unchanged

B1 Main calls AddObstaclesToColumn(3) only when simulation 2 is chosen

Test (13.2, 1 mark): inputs 2, 5, 5, 1 – screen capture showing ants never entering column 3

14 [12 marks]

Soldier ants (SoldierAnt, SetUpANestAt, AdvanceStage).

```

class SoldierAnt(Ant):
    def __init__(self, ...):
        super().__init__(...)
        self._AntType = "soldier"                # type of ant = soldier

    def ChooseCellToMoveTo(self, ...):            # overrides Ant.ChooseCellToMoveTo
        while True:
            NewCell = self.ChooseRandomNeighbour(...)
            if abs(NewCell.Row - HomeNest.Row) <= 2 and abs(NewCell.Column - HomeNest.Column) <= 2:
                break                               # within two cells of home nest
            self.ChangeCell(NewCell)

# Simulation.SetUpANestAt (when more than one nest)
if self.__NoOfNests > 1:
    for _ in range(2):
        self.__AddAnt(SoldierAnt(...), Row, Column)    # two soldier ants per nest

# Simulation.AdvanceStage – soldier attack
for Soldier in all soldier ants:
    for OtherAnt in ants in the same cell as Soldier:
        if OtherAnt.NestID != Soldier.NestID:        # ant from a different nest
            if random.random() < 0.9:                 # 90% chance
                destroy / remove OtherAnt

```

Mark scheme

B1 SoldierAnt class declared inheriting from Ant **B1** Constructor sets type to soldier (calling the parent constructor)

B1 ChooseCellToMoveTo overridden **B1** Calls ChooseRandomNeighbour **B1** Repeats until the chosen cell is within 2 cells of the home nest (both row and column difference ≤ 2)

B1 Uses ChangeCell to move the ant

B1 SetUpANestAt: condition for more than one nest **B1** Creates two soldier ants in each nest

B1 AdvanceStage: finds soldier ants and other ants in the same cell **B1** Only ants from other nests are attacked **B1** 90% chance of destruction using random number; ant removed

Test (14.2, 1 mark): inputs 4, 3, 2, 4 – screen capture showing soldier ants created in each nest