2.1 Elements of Computational Thinking: 1-hour Pack

OCR H446 · Calculators not allowed · about 60 minutes

Total Marks

Name: _____ Date: ____ / ____ / _____

Answer ALL questions. Show all your working.

1 Mr Whitfield stores the exam scores for five students in an array called scores, with indices 0 to 4. The initial contents are scores = [67, 45, 89, 23, 71]. He sorts them using the following algorithm:

```
FOR i <- 0 TO 3
  FOR j <- 0 TO 3 - i
    IF scores[j] > scores[j + 1] THEN
      temp <- scores[j]
      scores[j] <- scores[j + 1]
      scores[j + 1] <- temp
    ENDIF
  NEXT j
NEXT i
```

(a) Complete a trace table showing the contents of the array scores after each comparison (each value of j) during the first pass of the outer loop, i.e. when i = 0. **(4)**

(b) State how many comparisons of the IF condition take place during this first pass (i = 0), for this 5 item list. **(2)**

(c) State an expression, in terms of n, for the total number of comparisons of the IF condition made in the worst case when sorting a list of n items using this algorithm. **(2)**

(d) The inner loop bound (3 - i) means that, as i increases, fewer comparisons are made in each successive pass. Explain why this is a valid optimisation of the algorithm. **(2)**

(Total for Question 1 is 10 marks)

© Revision Library | revisionlibrary.co.uk

2 Elodie is optimising a program that calculates Fibonacci numbers, where Fibonacci(0) = 0, Fibonacci(1) = 1, and Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2) for n ≥ 2. She notices that a simple recursive version recalculates the same values many times over, so she decides to store (cache) results as they are calculated for reuse later, an example of the 'thinking ahead' computational thinking technique.

(a) Define what is meant by memoisation, and state which computational thinking technique it is an example of. **(2)**

(b) The bottom-up algorithm below calculates Fibonacci(n) by filling an array called cache: **(3)**

```
FUNCTION Fibonacci(n)
  DECLARE cache : ARRAY[0:n] OF INTEGER
  cache[0] <- 0
  cache[1] <- 1
  FOR i <- 2 TO n
    cache[i] <- cache[i - 1] + cache[i - 2]
  NEXT i
  RETURN cache[n]
ENDFUNCTION
```

Complete a trace table showing the value stored in cache[i] for each value of i from 2 to 6, when Fibonacci(6) is called.

(c) Rewrite Fibonacci as a recursive function, in OCR Exam Reference Language, that uses memoisation to avoid recomputing values. You should use a global array called cache, with every element initially set to -1 to show that no value has yet been calculated. **(5)**

(d) State one other situation in computing, other than calculating Fibonacci numbers, where caching improves performance, and explain why it helps. **(2)**

(Total for Question 2 is 12 marks)

© Revision Library | revisionlibrary.co.uk

3 Priya drives a delivery van and must visit customers at four locations, A, B, C and D, before returning to the depot P. The table below shows the distance, in km, between each pair of locations:
P-A=4, P-B=9, P-C=7, P-D=6, A-B=5, A-C=8, A-D=3, B-C=2, B-D=10, C-D=6.
Because checking every possible route becomes impractical as the number of locations grows, Priya's route-planning app uses the nearest-neighbour heuristic: starting at the depot, always travel next to the closest location not yet visited.

Distance matrix (km)

	P	A	B	C	D
P	-	4	9	7	6
A	4	-	5	8	3
B	9	5	-	2	10
C	7	8	2	-	6
D	6	3	10	6	-

(a) State why an exact algorithm that checks every possible route becomes impractical as the number of delivery locations increases. **(2)**

(b) Apply the nearest-neighbour heuristic, starting and ending at the depot P, to find a route visiting A, B, C and D. State the order in which locations are visited and calculate the total distance travelled. **(4)**

(c) Show that the route P-A-B-C-D-P is shorter than the nearest-neighbour route found in part (b), and explain what this shows about the nearest-neighbour heuristic. **(3)**

(d) Give one advantage and one disadvantage of using a heuristic approach, such as nearest-neighbour, rather than an exact algorithm for this kind of real-world routing problem. **(2)**

(Total for Question 3 is 11 marks)

© Revision Library | revisionlibrary.co.uk

4 Jayden is developing a funds transfer feature for Nexbank's mobile app. Two processes, ProcessA and ProcessB, may run at the same time and both need exclusive access to a shared balanceRecord and a shared transactionLog to complete a transfer:

```
ProcessA:
LOCK balanceRecord
LOCK transactionLog
// transfer funds
UNLOCK transactionLog
UNLOCK balanceRecord
ProcessB:
LOCK transactionLog
LOCK balanceRecord
// transfer funds
UNLOCK balanceRecord
UNLOCK transactionLog
```

(a) Define what is meant by 'thinking concurrently' as a computational thinking technique. **(2)**

(b) Explain, with reference to the pseudocode above, how a deadlock could occur if ProcessA and ProcessB run at the same time. **(3)**

(c) Suggest a change to the order of the LOCK statements in ProcessB that would prevent this deadlock, and explain why your change works. **(3)**

(d) State one difference between a deadlock and a race condition. **(2)**

(Total for Question 4 is 10 marks)

© Revision Library | revisionlibrary.co.uk

5 TransitLink is a logistics company whose route-planning software must decide the delivery order for up to 200 stops per van each day. Discuss how a combination of computational thinking techniques, decomposition, abstraction, heuristics and performance modelling, could be applied by TransitLink's developers to produce a workable, efficient solution, and evaluate the trade-offs involved in using approximate (heuristic) methods rather than exact algorithms for a problem of this scale.

(Total for Question 5 is 9 marks)

© Revision Library | revisionlibrary.co.uk

Mark scheme · 2.1 Elements of Computational Thinking: 1-hour Pack

Question 1

(a) j=0: correctly swaps to give [45, 67, 89, 23, 71] **B1**

(a) j=1: no swap, array unchanged [45, 67, 89, 23, 71] **B1**

(a) j=2: correctly swaps to give [45, 67, 23, 89, 71] **B1**

(a) j=3: correctly swaps to give [45, 67, 23, 71, 89] **B1**

(a) Answer: After i=0: [45, 67, 23, 71, 89]

(b) cao 4 **B1**

(b) correct reason, e.g. j takes the 4 values 0, 1, 2 and 3 so the IF condition is tested 4 times (oe, ft) **B1**

(b) Answer: 4 comparisons

(c) recognises the pattern that pass i makes (n-1-i) comparisons, summed over i=0 to n-2 (oe) **M1**

(c) cao n(n-1)/2 (oe equivalent forms accepted) **A1**

(c) Answer: n(n-1)/2

(d) recognises that after each pass, the largest remaining unsorted value has been moved into its correct position at the end of the unsorted section (oe) **B1**

(d) recognises that this position never needs to be compared again, so reducing the range of j reduces the total number of comparisons/improves efficiency without affecting correctness (oe) **B1**

(d) Answer: Each pass guarantees the largest unsorted value bubbles into its correct final position, so that position can be excluded from all later passes, reducing the total comparisons made without changing the result.

Question 2

(a) defines memoisation as storing/caching the results of expensive or repeated calculations so they can be reused instead of recalculated (oe) **B1**

(a) identifies 'thinking ahead' as the computational thinking technique **B1**

(a) Answer: Memoisation is the storing of the results of calculations so that, if the same calculation is needed again, the stored result can be reused instead of being recalculated. It is an example of thinking ahead.

(b) cache[2]=1 and cache[3]=2 both correct **B1**

(b) cache[4]=3 and cache[5]=5 both correct **B1**

(b) cao cache[6]=8 **B1**

(b) Answer: cache[2]=1, cache[3]=2, cache[4]=3, cache[5]=5, cache[6]=8

(c) any correct logic: cache array declared/initialised so that every element holds -1 before the function is first called **M1**

(c) correct base case: IF n ≤ 1 THEN RETURN n (oe) **M1**

(c) correct check: IF cache[n] <> -1 THEN RETURN cache[n] (or equivalent test that a value has already been stored) **M1**

(c) correct recursive calculation cache[n] <- Fibonacci(n - 1) + Fibonacci(n - 2) when no cached value exists **M1**

(c) correct RETURN cache[n] after the value has been stored; ignore minor syntax errors throughout **M1**

(c) Answer:

```
FOR i <- 0 TO n
  cache[i] <- -1
NEXT i

FUNCTION Fibonacci(n)
  IF n <= 1 THEN
    RETURN n
  ENDIF
  IF cache[n] <> -1 THEN
    RETURN cache[n]
  ENDIF
  cache[n] <- Fibonacci(n - 1) + Fibonacci(n - 2)
  RETURN cache[n]
ENDFUNCTION
```

(d) valid example, e.g. a web browser caching images/pages, or a CPU cache storing recently used memory values (oe) **B1**

(d) valid reason linked to the example, e.g. avoids the slower process of re-downloading/re-fetching the data from its original, slower source, reducing loading time/improving performance (oe, ft) **B1**

(d) Answer: e.g. a web browser caches images so that revisiting a page does not require re-downloading them, reducing loading time, because reading from the local cache is much faster than re-fetching data across a network.

© Revision Library | revisionlibrary.co.uk

Question 3

(a) identifies that the number of possible routes grows factorially as locations are added (oe) **B1**

(a) identifies that this makes checking every route computationally infeasible/take too long for even a moderate number of locations (oe) **B1**

(a) Answer: The number of possible routes grows factorially as more locations are added, so checking every route becomes computationally infeasible (takes too long) even for a moderate number of locations.

(b) correct method: at each step, selects the nearest unvisited location from the current location **M1**

(b) correct order of visits P-A-D-C-B-P **A1**

(b) correct individual distances used (4, 3, 6, 2, 9) **A1**

(b) cao total = 24 km **A1**

(b) Answer: P - A - D - C - B - P, total distance = 24 km

(c) correctly sums the distances for P-A-B-C-D-P: 4+5+2+6+6 **M1**

(c) cso 23 km, correctly identified as less than the 24 km found in part (b) **A1**

(c) explains that this shows the nearest-neighbour heuristic does not guarantee finding the optimal (shortest) route, only a reasonable one found quickly (oe) **B1**

(c) Answer: P-A-B-C-D-P = 4+5+2+6+6 = 23 km, which is less than the 24 km nearest-neighbour route. This shows that the nearest-neighbour heuristic does not always find the optimal route.

(d) valid advantage, e.g. produces a usable route very quickly, even for large numbers of locations, unlike an exact algorithm (oe) **B1**

(d) valid disadvantage, e.g. does not guarantee the optimal (shortest) route will be found, as shown in part (c) (oe) **B1**

(d) Answer: Advantage: it finds a workable route very quickly, even as the number of delivery locations grows. Disadvantage: it does not guarantee the shortest possible route will be found.

Question 4

(a) identifies that thinking concurrently involves considering tasks/processes that may run at the same time (oe) **B1**

(a) identifies that it involves designing an algorithm/system so that these simultaneous tasks interact correctly, e.g. share resources safely (oe) **B1**

(a) Answer: Thinking concurrently means designing an algorithm or system with the understanding that multiple tasks or processes may be running/executing at the same time, and ensuring they interact correctly, for example when sharing resources.

(b) identifies that ProcessA locks balanceRecord while ProcessB locks transactionLog at around the same time **M1**

(b) identifies that ProcessA then waits for transactionLog (held by ProcessB), while ProcessB waits for balanceRecord (held by ProcessA) **M1**

(b) cao concludes that neither process can proceed, so a deadlock (circular wait) occurs **A1**

(b) Answer: ProcessA locks balanceRecord while ProcessB locks transactionLog at almost the same time. ProcessA then waits to lock transactionLog, which ProcessB is holding, while ProcessB waits to lock balanceRecord, which ProcessA is holding. Neither process can proceed, so a deadlock occurs.

(c) correctly suggests reordering ProcessB to LOCK balanceRecord before LOCK transactionLog (i.e. the same order as ProcessA) **B1**

(c) explains that both processes now request the shared resources in the same order, so whichever process locks balanceRecord first will be able to acquire transactionLog before the other process can request it (oe) **B1**

(c) further valid development, e.g. identifies this as an example of the general deadlock-prevention technique of enforcing a consistent lock ordering/hierarchy (oe) **B1**

(c) Answer: Reorder ProcessB to LOCK balanceRecord then LOCK transactionLog, the same order as ProcessA. Because both processes now request the two resources in the same order, one process will always obtain both locks in turn without the other holding one it needs, removing the circular wait. This is a standard deadlock-prevention technique: enforcing a consistent lock ordering.

(d) correctly describes deadlock, e.g. two or more processes are permanently blocked, each waiting for a resource held by another (oe) **B1**

(d) correctly describes a race condition, e.g. processes do complete, but the final result depends unpredictably on the timing/order in which they access shared data, and may be incorrect/inconsistent (oe) **B1**

(d) Answer: In a deadlock, two or more processes are permanently blocked, each waiting for a resource held by the other, so none can proceed. In a race condition, the processes do complete, but the outcome depends unpredictably on the exact timing/order of access to shared data, which can produce an incorrect or inconsistent result.

© Revision Library | revisionlibrary.co.uk

Question 5

Level 1 Limited or generic discussion, largely descriptive rather than applied to the TransitLink scenario; little or no evaluation of the heuristic versus exact trade-off; technical terminology used inconsistently. **1-3**

Level 2 Reasonable discussion of at least two computational thinking techniques, applied with some success to the scenario; some evaluation of the heuristic versus exact trade-off, though it may be one-sided or underdeveloped; mostly appropriate terminology. **4-6**

Level 3 Detailed, coherent discussion of at least three computational thinking techniques, accurately applied to the TransitLink scenario; balanced evaluation of the trade-offs between heuristic and exact approaches, reaching a justified conclusion; technical terminology used accurately throughout. **7-9**

Indicative content:

- Decomposition: the overall route-planning problem can be broken down into smaller sub-problems, e.g. first grouping stops by depot/geographic zone, then deciding the delivery order within each zone.
- Abstraction: the algorithm can ignore irrelevant detail (e.g. individual driver preferences, exact parcel contents) while retaining the essential detail needed to plan a route (e.g. delivery time windows, road distances/times between stops).
- Heuristics: an exact algorithm checking every possible order of up to 200 stops is computationally infeasible, since the number of possible routes grows factorially; a heuristic (e.g. nearest-neighbour, or an improvement heuristic such as 2-opt) can produce a good, workable route in a practical amount of time.
- Performance modelling: a simulation of proposed routes (e.g. under varying traffic conditions or van breakdowns) can be used to test how well a route plan is likely to perform in practice before it is used operationally, without the cost/risk of testing on real deliveries.
- Evaluation of trade-off: heuristic methods are fast and scale well to large numbers of stops, but do not guarantee an optimal (shortest/cheapest) route; exact algorithms guarantee optimality but become impractical well before 200 stops are reached, so a combination of techniques (decomposition to reduce problem size, then heuristics within each sub-problem, refined and tested by performance modelling) is an appropriate, justified real-world solution given the operational need for near real-time route generation.

© Revision Library | revisionlibrary.co.uk