2.3 Algorithms: Complexity, Searching and Sorting

OCR H446 · Calculators not allowed · about 95 minutes · Write-in edition

Total Marks

Name: _____ Date: ____ / ____ / _____

Answer ALL questions. Show all your working.

1 A stock-control system for a small hardware shop in Leeds stores unsold item codes in an array called stock. The array is not sorted. All arrays in this paper are indexed from 0, so an array with n items has indices 0 to n - 1. Pseudocode follows OCR's Exam Reference Language conventions (assignment <-, iteration FOR ... NEXT / WHILE ... ENDWHILE / DO ... UNTIL, selection IF ... ENDIF).

```
FUNCTION LinearSearch(stock, target) RETURNS INTEGER
  FOR i <- 0 TO LEN(stock) - 1
    IF stock[i] = target THEN
      RETURN i
    ENDIF
  NEXT i
  RETURN -1
ENDFUNCTION
```

stock = [12, 7, 19, 3, 25, 8]

(a) Complete the trace table below to show LinearSearch(stock, 25) being called. Add a row for each value of i tested, showing stock[i] and whether stock[i] = target is true or false. **(5)**

i	stock[i]	stock[i] = target?
0		
1		
2		
3		
4		

(b) State the value returned by the function call in part (a). **(1)**

(c) The array is now searched for the value 30, which is not present. State the number of times the condition stock[i] = target is tested, and give the value returned by the function. **(2)**

(d) State, using Big O notation, the worst-case time complexity of linear search on a list of n items. **(1)**

(Total for Question 1 is 9 marks)

2 A school's exam-results system stores candidate numbers in ascending order in the array numbers. Binary search can be used because the array is sorted.

numbers = [3, 7, 12, 15, 19, 23, 28, 34, 41, 50] (indices 0 to 9)

```
FUNCTION BinarySearch(numbers, target) RETURNS INTEGER
  low <- 0
  high <- LEN(numbers) - 1
  WHILE low <= high
    mid <- (low + high) DIV 2
    IF numbers[mid] = target THEN
      RETURN mid
    ELSE IF numbers[mid] < target THEN
      low <- ______
    ELSE
      high <- ______
    ENDIF
  ENDWHILE
  RETURN -1
ENDFUNCTION
```

(a) Write the missing pseudocode for the two blank lines above, so that the algorithm correctly narrows the search range. **(2)**

(b) Complete the trace table below for the call BinarySearch(numbers, 34). **(4)**

low	high	mid	numbers[mid]	result of comparison

(c) State the index returned, and the number of times numbers[mid] is compared with the target. **(2)**

(d) Each element of numbers is stored using 2 bytes, and the array starts at hexadecimal memory address 0x4000. Calculate the hexadecimal memory address of the element found in part (c). **(2)**

(e) Give the 8-bit binary representation of the byte offset (14) used in the calculation in part (d). **(1)**

(Total for Question 2 is 11 marks)

3 A car park's number-plate recognition system checks whether a scanned number plate matches one already stored in a list.

(a) The list of stored number plates is not kept in any particular order. Explain why binary search could not be used to search this list directly, and state one thing that would need to be done first if binary search were to be used. **(2)**

(b) The car park has 1000 stored number plates. Calculate the maximum number of comparisons linear search would need to make to determine that a scanned plate is not in the list. **(1)**

(c) Once sorted, the maximum number of comparisons binary search needs for a list of n items can be found using ceil(log2(n + 1)). Calculate the maximum number of comparisons needed to search the sorted list of 1000 number plates. **(2)**

(d) State, giving a reason based on your answers to (b) and (c), which search algorithm is more time-efficient for this large list. **(2)**

(Total for Question 3 is 7 marks)

4 A DIY store in Cardiff records the prices (in pounds) of five items on a shelf label strip in the order [5, 2, 9, 1, 6]. They are sorted using the bubble sort algorithm below.

```
PROCEDURE BubbleSort(list)
  n <- LEN(list)
  DO
    swapped <- FALSE
    FOR i <- 0 TO n - 2
      IF list[i] > list[i + 1] THEN
        temp <- list[i]
        list[i] <- list[i + 1]
        list[i + 1] <- temp
        swapped <- TRUE
      ENDIF
    NEXT i
    n <- n - 1
  UNTIL swapped = FALSE
ENDPROCEDURE
```

(a) Explain the purpose of the swapped flag in the pseudocode above. **(2)**

(b) Complete the trace table below for BubbleSort([5, 2, 9, 1, 6]), showing the contents of the list, the value of swapped, and the value of n at the end of each full pass. **(6)**

pass	list at end of pass	swapped	n
1			
2			
3			
4			

(c) State the total number of comparisons and the total number of swaps made during the whole sort in part (b). **(2)**

(d) Explain how the algorithm knows to stop after pass 4 rather than continuing to check further passes. **(2)**

(Total for Question 4 is 12 marks)

5 A leisure centre stores four booking reference numbers [8, 3, 6, 1] which must be sorted into ascending order using insertion sort.

```
FUNCTION InsertionSort(list)
  FOR i <- 1 TO LEN(list) - 1
    current <- list[i]
    j <- i - 1
    WHILE j >= 0 AND list[j] > current
      ______
      j <- j - 1
    ENDWHILE
    list[j + 1] <- current
  NEXT i
ENDFUNCTION
```

(a) Using the pseudocode above, explain how insertion sort builds up the sorted part of the list as i increases. **(2)**

(b) Complete the trace table below for InsertionSort([8, 3, 6, 1]), showing the value of current and the list contents at the end of each iteration of the outer FOR loop (i = 1, 2, 3). **(6)**

i	current	list at end of this iteration
1		
2		
3		

(c) State the final sorted list produced. **(1)**

(d) Write the missing line of pseudocode (marked _____) needed inside the WHILE loop so the algorithm shifts elements correctly. **(2)**

(Total for Question 5 is 11 marks)

6 A playlist of 8 song lengths (in seconds) needs to be sorted using merge sort: [230, 180, 340, 95, 410, 150, 275, 60].

(a) Describe what happens during the divide stage and the merge (conquer) stage when merge sort sorts this playlist. **(3)**

(b) State how many times the playlist of 8 songs must be split before every sub-list contains a single element. **(1)**

(c) Explain why merge sort has a time complexity of O(n log n). **(3)**

(Total for Question 6 is 7 marks)

7 A library keeps an array titles of book titles sorted alphabetically. When a new book arrives, the system needs to find the position where the new title should be inserted so that titles stays in alphabetical order. If the title is already present, the function should return its existing index instead of inserting a duplicate.

(a) Write a function FindInsertPosition(titles, newTitle) RETURNS INTEGER, using a binary-search approach (rather than checking every element in turn), that: **(6)**
- returns the index of newTitle if it is already present in titles, or
- otherwise returns the index at which newTitle should be inserted to keep the array sorted.

You may assume string comparisons (e.g. titles[mid] < newTitle) compare alphabetical order correctly. Use OCR pseudocode conventions. Marks are awarded for correct logic; minor syntax errors will be ignored.

(Total for Question 7 is 6 marks)

8 The table below lists five algorithms covered in this topic.

(a) Complete the table below by writing the worst-case time complexity of each algorithm, using Big O notation. **(5)**

Algorithm	Worst-case time complexity
Linear search	
Binary search	
Bubble sort	
Insertion sort	
Merge sort	

(b) State, for each of bubble sort, insertion sort and merge sort, whether it requires significant additional (auxiliary) memory beyond the original list, or whether it can sort in place using only a small, constant amount of extra memory. **(3)**

(c) A shop assistant has a list of 20 till receipts that are already almost fully sorted by value, with only two receipts out of place. State, with a justification, which of bubble sort or insertion sort would typically perform fewer operations on this nearly-sorted list. **(2)**

(Total for Question 8 is 10 marks)

9 A supermarket's loyalty-card database holds 2,000,000 customer records, sorted by customer ID. Throughout the day, staff repeatedly need to search for a customer's record by ID. Every night, a batch of new customer records is added, and the whole file must then be back in ID order ready for the next day's searching.

(a) Discuss which searching algorithm and which sorting algorithm would be most appropriate for this system, justifying your recommendation by referring to the time complexity of the algorithms you have studied in this topic. **(6)**

(Total for Question 9 is 6 marks)

Mark scheme · 2.3 Algorithms: Complexity, Searching and Sorting

Question 1

(a) i=0: stock[0]=12, false **B1**

(a) i=1: stock[1]=7, false **B1**

(a) i=2: stock[2]=19, false **B1**

(a) i=3: stock[3]=3, false **B1**

(a) i=4: stock[4]=25, true (loop stops, RETURN 4) **B1**

(a) Answer: i=0 (12,F), i=1 (7,F), i=2 (19,F), i=3 (3,F), i=4 (25,T) -> returns 4

(b) 4 ft from (a) **B1**

(b) Answer: 4

(c) 6 comparisons made (every element is tested since the target is absent) **B1**

(c) returns -1 **B1**

(c) Answer: 6 comparisons are made; the function returns -1.

(d) O(n) **B1**

(d) Answer: O(n)

Question 2

(a) low <- mid + 1 **B1**

(a) high <- mid - 1 **B1**

(a) Answer: low <- mid + 1 ; high <- mid - 1

(b) row 1: low=0, high=9, mid=4 **B1**

(b) row 1: numbers[4]=19, 19 < 34 so low <- 5 **B1**

(b) row 2: low=5, high=9, mid=7 **B1**

(b) row 2: numbers[7]=34, 34 = 34 so RETURN 7 **B1**

(b) Answer: Row1: low=0,high=9,mid=4,numbers[4]=19 (< target, low<-5). Row2: low=5,high=9,mid=7,numbers[7]=34 (= target, return 7).

(c) index 7 ft from (b) **B1**

(c) 2 comparisons ft from (b) **B1**

(c) Answer: Index 7 is returned; 2 comparisons are made.

(d) address = base + (index x size) = 0x4000 + (7 x 2) **M1**

(d) 0x400E cao **A1**

(d) Answer: 0x400E

(e) 00001110 **B1**

(e) Answer: 00001110

Question 3

(a) binary search requires the data to be sorted (in order), as it relies on discarding half the remaining items each time (oe) **B1**

(a) the list would need to be sorted first (e.g. using a sorting algorithm) before binary search could be applied **B1**

(a) Answer: Binary search needs the list to be in order so it can repeatedly halve the search space; an unsorted list would first need to be sorted before binary search could be used.

(b) 1000 **B1**

(b) Answer: 1000

(c) log2(1001) = awrt 9.97 **M1**

(c) ceil(9.97) = 10 cao **A1**

(c) Answer: 10

(d) binary search (oe) **B1**

(d) because it needs far fewer comparisons (10 compared with 1000), as O(log n) grows far more slowly than O(n) as n increases (oe) **B1**

(d) Answer: Binary search, because it needs only 10 comparisons in the worst case compared with 1000 for linear search, reflecting O(log n) growing much more slowly than O(n).

Question 4

(a) swapped is set to TRUE whenever a swap is made during a pass **B1**

(a) if no swaps occur during a pass, the list must already be sorted, so the algorithm can stop early (avoiding unnecessary further passes) **B1**

(a) Answer: It allows the algorithm to detect when a full pass makes no swaps, meaning the list is already sorted, so it can stop instead of running unnecessary further passes.

(b) pass 1: list = [2, 5, 1, 6, 9] **B1**

(b) pass 2: list = [2, 1, 5, 6, 9] **B1**

(b) pass 3: list = [1, 2, 5, 6, 9] **B1**

(b) pass 4: list = [1, 2, 5, 6, 9] (unchanged) **B1**

(b) swapped correctly shown as TRUE, TRUE, TRUE, FALSE for passes 1-4 **B1**

(b) n correctly shown decreasing: 4, 3, 2, 1 at the end of passes 1-4 **B1**

(b) Answer: Pass1 [2,5,1,6,9] swapped=T n=4; Pass2 [2,1,5,6,9] swapped=T n=3; Pass3 [1,2,5,6,9] swapped=T n=2; Pass4 [1,2,5,6,9] swapped=F n=1

(c) 10 comparisons ft **B1**

(c) 5 swaps ft **B1**

(c) Answer: 10 comparisons; 5 swaps.

(d) on pass 4 no elements are out of order, so no swaps take place and swapped remains FALSE **B1**

(d) the DO...UNTIL loop condition (UNTIL swapped = FALSE) is then met, so the loop, and the algorithm, ends **B1**

(d) Answer: In pass 4 the list is already fully sorted, so no swaps occur and swapped stays FALSE; this satisfies the UNTIL swapped = FALSE condition, ending the loop.

Question 5

(a) at each value of i, the element current = list[i] is removed and compared backwards against the already-sorted elements to its left **B1**

(a) elements greater than current are shifted one place to the right until the correct position is found, and current is then inserted there **B1**

(a) Answer: For each i, the element current is taken out and compared with the sorted elements before it; any elements greater than current are shifted right until the correct gap is found, and current is inserted there.

(b) i=1: current = 3 **B1**

(b) i=1: list = [3, 8, 6, 1] **B1**

(b) i=2: current = 6 **B1**

(b) i=2: list = [3, 6, 8, 1] **B1**

(b) i=3: current = 1 **B1**

(b) i=3: list = [1, 3, 6, 8] **B1**

(b) Answer: i=1: current=3, list=[3,8,6,1]; i=2: current=6, list=[3,6,8,1]; i=3: current=1, list=[1,3,6,8]

(c) [1, 3, 6, 8] ft from (b) **B1**

(c) Answer: [1, 3, 6, 8]

(d) shifts the element at position j one place to the right **M1**

(d) list[j + 1] <- list[j] (or equivalent correct logic; ignore minor syntax) **A1**

(d) Answer: list[j + 1] <- list[j]

Question 6

(a) divide: the list is repeatedly split into two halves until each sub-list contains only one element (8 -> two lists of 4 -> four lists of 2 -> eight lists of 1) **B1**

(a) merge: pairs of sorted sub-lists are combined by repeatedly comparing the front elements of each sub-list and taking the smaller first **B1**

(a) merging continues, combining larger sorted sub-lists, until a single fully sorted list of all 8 elements remains **B1**

(a) Answer: Divide: the list is split in half repeatedly until each sub-list has one element. Merge: pairs of sorted sub-lists are combined by comparing their front elements, repeating until one sorted list of all 8 elements remains.

(b) 3 **B1**

(b) Answer: 3

(c) the list is halved repeatedly until sub-lists of size 1 remain, giving log2(n) levels of splitting/merging **B1**

(c) at each level, merging all the sub-lists back together requires a total of (up to) n comparisons across that level **B1**

(c) multiplying the number of levels (log n) by the work per level (n) gives an overall time complexity of O(n log n) **B1**

(c) Answer: There are log2(n) levels of splitting/merging, and each level does up to n comparisons in total; multiplying levels by work per level gives O(n log n).

Question 7

(a) initialises low <- 0 and high <- LEN(titles) - 1 **M1**

(a) uses a loop (e.g. WHILE low ≤ high) that computes mid <- (low + high) DIV 2 each iteration **M1**

(a) returns mid immediately if titles[mid] = newTitle **M1**

(a) correctly narrows the range: low <- mid + 1 if titles[mid] < newTitle, else high <- mid - 1 **M1**

(a) when the loop ends without finding a match, returns low as the correct insertion position **M1**

(a) overall algorithm is logically complete and correctly ordered (accept equivalent correct logic; ignore minor syntax errors) **A1**

(a) Answer:

```
FUNCTION FindInsertPosition(titles, newTitle) RETURNS
  low <- 0
  high <- LEN(titles) - 1
  WHILE low <= high
    mid <- (low + high) DIV 2
    IF titles[mid] = newTitle THEN
      RETURN mid
    ELSE IF titles[mid] < newTitle THEN
      low <- mid + 1
    ELSE
      high <- mid - 1
    ENDIF
  ENDWHILE
  RETURN low
ENDFUNCTION
```

Question 8

(a) Linear search: O(n) **B1**

(a) Binary search: O(log n) **B1**

(a) Bubble sort: $O(n^2)$ **B1**

(a) Insertion sort: $O(n^2)$ **B1**

(a) Merge sort: O(n log n) **B1**

(a) Answer: Linear search O(n); Binary search O(log n); Bubble sort $O(n^2)$; Insertion sort $O(n^2)$; Merge sort O(n log n).

(b) bubble sort: in place, O(1) extra memory **B1**

(b) insertion sort: in place, O(1) extra memory **B1**

(b) merge sort: requires significant additional memory, O(n) extra memory (for the temporary arrays used when merging) **B1**

(b) Answer: Bubble sort and insertion sort both sort in place (O(1) extra memory); merge sort needs O(n) extra memory for its temporary merge arrays.

(c) insertion sort **B1**

(c) because insertion sort only has to shift the small number of out-of-place elements a short distance, so on a nearly-sorted list it performs close to O(n) comparisons, whereas bubble sort's structure typically still leads to more comparisons before it can detect the list is sorted (oe, allow reasonable justification about adaptiveness) **B1**

(c) Answer: Insertion sort, because it adapts well to nearly-sorted data, only needing to shift the few out-of-place elements, giving close to O(n) performance rather than the larger number of comparisons typical of bubble sort.

Question 9

(a) **L1** Basic, isolated points are made about searching and/or sorting algorithms, with little or no reference to time complexity or to the given scenario. The answer may simply name algorithms with no justification. **1-2**

(a) **L2** Some relevant points are made about appropriate searching and sorting algorithms, with limited reference to time complexity (e.g. Big O notation used but not clearly applied to the scale of the data). Some link is made to the scenario (2,000,000 records, daily batch updates). **3-4**

(a) **L3** A clear, well-justified discussion is given, correctly identifying appropriate searching and sorting algorithms, supported by accurate use of time complexity (Big O notation) applied specifically to the scale of the scenario (2,000,000 records) and the fact that the data is sorted and updated in daily batches. **5-6**

(a) Indicative content:

- Binary search is appropriate for the daytime searches because the file is kept sorted; its O(log n) complexity means even for 2,000,000 records only around 21 comparisons are needed (log2(2,000,000) is approximately 21), compared with up to 2,000,000 for linear search O(n).
- A simple O(n^2) algorithm such as bubble sort or insertion sort would be far too slow to re-sort 2,000,000 records overnight, since n^2 for n = 2,000,000 is an enormous number of operations.
- Merge sort, with worst-case time complexity O(n log n), is a much more suitable choice for re-sorting the large file each night, as it scales far better than an O(n^2) algorithm for large n.
- Merge sort does require additional (auxiliary) memory, O(n), for the temporary arrays used while merging, which is a cost worth mentioning, but this is an acceptable trade-off given the huge time saving.
- A stronger answer may note that, because the file is already sorted and only a batch of new records needs adding, an efficient approach could merge the (small) sorted batch of new records into the (large) already-sorted file, which is itself essentially a merge-sort merge step, rather than re-sorting everything from scratch.
- Overall recommendation: binary search for searching (O(log n)) and merge sort for the nightly re-sort (O(n log n)), justified by the scale of the data (2,000,000 records) making O(n) or O(n^2) approaches impractical.

(a) Answer: Best answers recommend binary search (O(log n)) for daily lookups and merge sort (O(n log n)) for the nightly re-sort, justified using Big O notation applied to n = 2,000,000.