2.3 Algorithms: Complexity, Searching and Sorting: 30-minute Pack

OCR H446 · Calculators not allowed · about 30 minutes

Total Marks

Name: _____ Date: ____ / ____ / _____

Answer ALL questions. Show all your working.

1 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 1 is 11 marks)

© Revision Library | revisionlibrary.co.uk

2 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 2 is 7 marks)

© Revision Library | revisionlibrary.co.uk

3 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 3 is 7 marks)

---

4 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 4 is 6 marks)

© Revision Library | revisionlibrary.co.uk

Mark scheme · 2.3 Algorithms: Complexity, Searching and Sorting: 30-minute Pack

Question 1

(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 2

(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 3

(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 4

(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
```


© Revision Library | revisionlibrary.co.uk