2.6 Advanced Algorithms: Searching, Sorting and Path-Finding: Test Pack

OCR H446 · Calculators not allowed · about 40 minutes

Total Marks

Name: _____ Date: ____ / ____ / _____

Answer ALL questions. Show all your working.

1 A vintage record shop in Brighton stores its catalogue of vinyl records as an array of catalogue numbers, kept sorted in ascending order. 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, selection IF ... ENDIF).

catalogueNumbers = [102, 118, 125, 130, 144, 151, 168, 172, 190, 205, 211, 230] (indices 0 to 11)

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

(a) State one precondition that must be true about an array before binary search can be used on it correctly. **(1)**

(b) Complete the trace table below for the call BinarySearch(catalogueNumbers, 172). **(4)**

low	high	mid	list[mid]	comparison result

(c) State the value returned by the function call, and the number of comparisons of list[mid] with the target that were made. **(2)**

(d) The shop's catalogue grows to 2000 records, kept sorted. Using the formula ceil(log2(n + 1)) for the maximum number of comparisons binary search needs on a sorted list of n items, calculate the maximum number of comparisons needed to search the 2000-record catalogue. **(2)**

(e) Calculate the maximum number of comparisons linear search would need on the same 2000-record catalogue if a scanned record is not present, and state, with a reason based on Big O notation, which search algorithm is more time-efficient for this catalogue. **(3)**

(Total for Question 1 is 12 marks)

© Revision Library | revisionlibrary.co.uk

2 Both bubble sort and insertion sort can be optimised to recognise when a list is fully sorted and stop early. The list [1, 2, 3, 8, 4, 5] is nearly sorted: only the value 8 is out of place.

```
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

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

(a) Complete the trace table below for BubbleSort([1, 2, 3, 8, 4, 5]), showing the list contents, whether any swap occurred, at the end of each pass, and the total number of comparisons made in that pass. **(4)**

pass	list at end of pass	swapped	comparisons this pass
1			
2			

© Revision Library | revisionlibrary.co.uk

(b) Complete the trace table below for InsertionSort([1, 2, 3, 8, 4, 5]), showing the value of current and the number of comparisons made against list[j] for each value of i. (For i = 1, 2 and 3, current is already in the correct place and no shifts occur.) **(4)**

i	current	comparisons made	list at end of this iteration
1	2	1	[1, 2, 3, 8, 4, 5]
2	3	1	[1, 2, 3, 8, 4, 5]
3	8	1	[1, 2, 3, 8, 4, 5]
4			
5			

(c) State the total number of comparisons made by each algorithm across the whole sort, and state which algorithm required fewer comparisons to sort this nearly-sorted list. **(3)**

(d) Explain, with reference to your answers to part (c), why insertion sort is often preferred over bubble sort when sorting data that is already nearly in order, even though both algorithms have the same $O(n^2)$ worst-case time complexity. **(2)**

(Total for Question 2 is 13 marks)

© Revision Library | revisionlibrary.co.uk

3 The table below covers the searching and sorting algorithms studied so far in this topic.

(a) Complete the table below, giving the worst-case time complexity and the best-case time complexity (assuming any early-exit optimisation is used) of each algorithm, using Big O notation. **(5)**

Algorithm	Worst-case	Best-case
Linear search		
Binary search		
Bubble sort		
Insertion sort		
Merge sort		

(b) State which of the five algorithms in the table are described as adaptive (perform noticeably better on data that is already sorted or nearly sorted than their worst case), and explain why merge sort does not have this property. **(3)**

(Total for Question 3 is 8 marks)

© Revision Library | revisionlibrary.co.uk

Mark scheme · 2.6 Advanced Algorithms: Searching, Sorting and Path-Finding: Test Pack

Question 1

(a) the array must already be sorted (in ascending or descending order) (oe) **B1**

(a) Answer: The array must already be sorted into order.

(b) row 1: low=0, high=11, mid=5, list[5]=151, 151 < 172 so low <- 6 **B1**

(b) row 2: low=6, high=11, mid=8, list[8]=190, 190 > 172 so high <- 7 **B1**

(b) row 3: low=6, high=7, mid=6, list[6]=168, 168 < 172 so low <- 7 **B1**

(b) row 4: low=7, high=7, mid=7, list[7]=172, 172 = 172 so RETURN 7 **B1**

(b) Answer: Row1: low0,high11,mid5(151,<,low<-6). Row2: low6,high11,mid8(190,>,high<-7). Row3: low6,high7,mid6(168,<,low<-7). Row4: low7,high7,mid7(172,=,return7).

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

(c) 4 comparisons ft from (b) **B1**

(c) Answer: 7 is returned; 4 comparisons are made.

(d) log2(2001) = awrt 10.97 **M1**

(d) ceil(10.97) = 11 cao **A1**

(d) Answer: 11

(e) 2000 **B1**

(e) binary search (oe) **B1**

(e) because it needs far fewer comparisons (11 compared with 2000), as O(log n) grows far more slowly than O(n) as n increases (oe) **B1**

(e) Answer: 2000 comparisons for linear search; binary search is far more time-efficient, needing only 11 comparisons compared with 2000, reflecting O(log n) growing much more slowly than O(n).

Question 2

(a) pass 1: list = [1, 2, 3, 4, 5, 8] **B1**

(a) pass 1: swapped = TRUE, comparisons = 5 **B1**

(a) pass 2: list = [1, 2, 3, 4, 5, 8] (unchanged) **B1**

(a) pass 2: swapped = FALSE, comparisons = 4 **B1**

(a) Answer: Pass1: [1,2,3,4,5,8], swapped=TRUE, 5 comparisons. Pass2: [1,2,3,4,5,8], swapped=FALSE, 4 comparisons.

(b) i=4: current = 4 **B1**

(b) i=4: 2 comparisons made, list = [1, 2, 3, 4, 8, 5] **B1**

(b) i=5: current = 5 **B1**

(b) i=5: 2 comparisons made, list = [1, 2, 3, 4, 5, 8] **B1**

(b) Answer: i=4: current=4, 2 comparisons, list=[1,2,3,4,8,5]. i=5: current=5, 2 comparisons, list=[1,2,3,4,5,8].

(c) bubble sort: 9 comparisons ft from (a) **B1**

(c) insertion sort: 1+1+1+2+2 = 7 comparisons ft from (b) **B1**

(c) insertion sort required fewer comparisons (7 compared with 9) **B1**

(c) Answer: Bubble sort: 9 comparisons. Insertion sort: 7 comparisons. Insertion sort needed fewer.

(d) insertion sort only performs comparisons where they are actually needed to find the correct insertion point for each element, so an element already in its correct position (like the first three here) costs only a single comparison **B1**

(d) bubble sort must still compare every adjacent pair on each full pass, regardless of how many elements are already in order, which cost it 2 more comparisons than insertion sort on this example (9 vs 7) **B1**

(d) Answer: Insertion sort only compares as far back as needed to find each element's correct position, so already-placed elements cost just one comparison each; bubble sort must still compare every adjacent pair on every pass regardless, which is why it needed more comparisons (9) than insertion sort (7) on this nearly-sorted list.

© Revision Library | revisionlibrary.co.uk

Question 3

(a) linear search: worst O(n), best O(1) **B1**

(a) binary search: worst O(log n), best O(1) **B1**

(a) bubble sort: worst $O(n^2)$, best O(n) **B1**

(a) insertion sort: worst $O(n^2)$, best O(n) **B1**

(a) merge sort: worst O(n log n), best O(n log n) **B1**

(a) Answer: Linear search: O(n)/O(1). Binary search: O(log n)/O(1). Bubble sort: $O(n^2)$/O(n). Insertion sort: $O(n^2)$/O(n). Merge sort: O(n log n)/O(n log n).

(b) linear search, binary search, bubble sort and insertion sort are all adaptive (their best case is better than their worst case) **B1**

(b) merge sort is not adaptive: its best case equals its worst case, O(n log n) **B1**

(b) because standard merge sort always fully splits the list down to single elements and always fully merges every sub-list back together, regardless of whether the input was already in order (oe) **B1**

(b) Answer: Linear search, binary search, bubble sort and insertion sort are all adaptive. Merge sort is not, because it always performs the same full sequence of splits and merges (O(n log n)) no matter how the input list is initially ordered.

© Revision Library | revisionlibrary.co.uk