A

CS2040 — AY2025/2026 Semester 1

Written Assessment 1 (Solutions)
45 minutes / 40 marks

Q1 [3 marks]

What is the tightest big-O time complexity of the function `func(arr, 0)` with respect to the length `n` of the array arr?

```
public static int func(int[] arr, int i) {
      int n = arr.length;
      if (i == n) {
            return 0;
      } else {
            int sum = 0;
            for (int j = 0; j < n; j++) {
                  sum += arr[j];
            }
            return sum + func(arr, i + 1);
      }
}
```

a. O(n) **b.** O(n^2) **(Ans)** **c.** O(nlogn) **d.** O($n^2 \cdot \log n$) **e.** O(n^3)
f. None of the above

Explanation: The function func has a recursion that runs n times, and at each recursion level, it executes a loop that also runs n times. Since the recursion depth is n and each level involves a loop of size n, the overall time complexity is O(n) multiplied by n, resulting in O(n2).

Q2 [3 marks]

What is the tightest big-O time complexity of the following program (assume n>1)?

```
int total = 0;
for (int i = 1; i <= n; i *= 2) {
      for (int j = 1; j <= n; j++) {
            for (int k = 0; k < j; k++) {
                  total++;
            }
      }
}
```

a. O(n^2) **b.** O(n log n) **c.** O(n^3) **d.** O(n^2 log n) **(Ans)** **e.** None of the above

Explanation:
The outermost loop increases i by multiplying it by 2 each time, so it runs about log n times. For each value of i, the middle loop runs n times. Inside the middle loop, the innermost loop executes roughly j steps. When we add up all the work for a fixed i, the total is in the order of n^2 operations. Since this happens for each of the log n values of i, the total running time grows on the order of n^2 log n.

Q3 [3 marks]

Consider a singly linked list with a head pointer only. Which of the following operations can always be performed in O(1) time?

I. Insert a new node at the beginning of the list

II. Insert a new node at the end of the list

III. Delete the first node of the list

IV. Delete the last node of the list

a. I and II only **b.** I and III only **(Ans)** **c.** II and IV only **d.** I, II, and III only
e. All of I, II, III, and IV

Explanation:
I. Insert at the beginning: We can create a new node, point it to the current head, and update the head pointer — all in O(1) time.

II. Insert at the end: Requires traversing the entire list to find the last node (since there's no tail pointer), which takes O(n) time.

III. Delete the first node: Simply update the head pointer to head.next — this is O(1).

IV. Delete the last node: Requires finding the second-to-last node (to set its next to null), which involves traversal — O(n) time.

Hence, only I and III can be done in O(1) time.

Q4 [3 marks]

A stack is initially empty. After a sequence of push and pop operations, which of the following sequences of operations results in element 10 being at the top of the stack in the final state?

I. Push 20 → Push 10 → Push 20 → Push 10 → Push 20 → Pop → Pop → Pop → Pop
II. Push 10 → Push 10→ Pop → Push 30 → Push 20→ Pop
III. Push 10 → Push 20 → Push 10 → Pop → Push 30 → Pop → Pop
IV. Push 10 → Push 20 → Push 30 → Pop → Pop
V. Push 10 → Push 10 → Pop → Push 20 → Pop → Pop

a. II and III **b.** I and IV **c.** III and IV**(Ans)** **d.** IV and V **e.** all of them

Explanation:
I: [20]→[20,10]→[20,10,20]→[20,10,20,10]→[20,10,20,10,20]→…→[20]

II. [10]→[10,10]→[10]→[10,30]→[10,30,20]→[10,30]

III: [10]→[10,20]→[10,20,10]→[10,20]→[10,20,30]→[10,20]→[10]

IV: [10]→[10,20]→[10,20,30]→[10,20]→[10]

V: [10]→[10,10]→[10]→[10,20]→[10]→[]

This Scenario will be used for **Q5** and **Q6.**

You are developing a library management system to keep track of books in a library. Each book has attributes such as its title, author, publication year, and the number of copies available. The system needs to sort all the books based on the number of copies available in descending order (from most copies to least). The sorting algorithm you choose uses the following logic:

1. Divide the list of books into two halves.
2. Sort each half recursively using this same algorithm.
3. Merge the sorted halves back together, ensuring that the books are in descending order based on the number of copies available.

Q5 [3 marks]
What is the time complexity of the sorting algorithm used to sort the books based on the number of copies available?

a. O(1) **b.** O(n) **c.** $O(n^2)$ **d.** O(n log n)**(Ans)** **e.** $O(n^2 \log n)$

f. None of the above

Explanation:
The sorting algorithm described follows the logic of **merge sort**, a **divide-and-conquer** algorithm. In merge sort, the list is divided into two halves recursively, which takes O(log n) time. At each level of recursion, the merging process involves combining two sorted sublists, which takes O(n) time. Since the merging process is repeated at every level of recursion, the overall time complexity is O(n log n), where n is the number of books in the library.

Q6 [3 marks] This is a continuation from Q5, make sure you attempt Q5 before Q6.

Considering the sorting algorithm described in Q5, what is the additional space complexity required by the algorithm to sort the students based on their grades?
(Exclude the space already used to store the book list itself.)

a. O(1) **b.** O(n) **(Ans)** **c.** $O(n^2)$ **d.** O(n log n) **e.** $O(n^2 \log n)$

f. None of the above

Explanation:

The algorithm uses merge sort, which requires extra space to store the sublists during the merging process. At each recursive level, it needs O(n) space to store the merged sublists. Since the space needed is proportional to the size of the list, the overall space complexity is O(n), where n is the number of books.

Q7 [3 marks]

A hash table with 13 slots (13 is a prime number) uses double hashing for collision resolution. The hash function is h(k) = k mod 13, and the secondary hash function for probing is h2(k) = 1 + (k mod 12). The table starts off empty, then keys 17, 30, and 43 are inserted in that order. After the 3 insertions, what is the final slot index where the key 43 is stored? Assume the table size never grows.

a. 4 **b.** 11 **c.** 12**(Ans)** **d.** 7 **e.** None of the above

Explanation:

The hash table has n = 13 slots, with double hashing using h(k) = k mod 13 as the primary hash function and $h_2(k)$ = 1 + (k mod 12) as the secondary hash function. The probe sequence for a key k is given by (h(k) + i * h_2(k)) mod 13, where i = 0, 1, 2, …. We insert the keys 17, 30, and 43 in order, and determine the final slot for key 43.

Key 17: h(17) = 17 mod 13 = 4. Slot 4 is empty, so 17 is placed in slot 4.

Key 30: h(30) = 30 mod 13 = 4. Slot 4 is occupied. Compute h_2(30) = 1 + (30 mod 12) = 1 + 6 = 7. Next probe: (4 + 7) mod 13 = 11. Slot 11 is empty, so 30 is placed in slot 11.

Key 43: h(43) = 43 mod 13 = 4. Slot 4 is occupied. Compute h_2(43) = 1 + (43 mod 12) = 1 + 7 = 8. First probe: (4 + 8) mod 13 = 12. Slot 12 is empty, so 43 is placed in slot 12.

Thus, the final slot for key 43 is 12.

Q8 [3 marks]

In a Min-Heap-based Priority Queue, which element is always removed first when performing a dequeue operation?

a. The smallest element **(Ans)** **b.** The largest element **c.** The last element added

d. A random element **e.** None of the above

Explanation:

In a Min-Heap-based Priority Queue, the heap property ensures that the smallest element is always at the root of the heap. The dequeue operation removes and returns the root element, which is the smallest element in the heap.

Q9 [5 marks]

Consider an array of length n that is strictly decreasing: A = [n, n−1, …, 2, 1].

Compare 2 sorting algorithms to sort A in increasing order:

- Standard Bubble Sort: exactly n−1 passes; adjacent compare-and-swap each pass.
- Optimized Bubble Sort: same as above but with a swapped flag; if a full pass has no swaps, terminate early. No other optimizations (no last-swap boundary, no bidirectional pass).

Claim: For this input, both algorithms have the same time complexity, namely $O(n^2)$ (and not better than that).

a. The claim is True/False. [2 marks] True
b. Give a brief justification (1–2 sentences). [3 marks]

In strictly decreasing order every pass performs swaps, so the early stop condition never triggers. Both run about n−1 passes with roughly n(n−1)/2 comparisons, which is $O(n^2)$.

Q10 [4 marks]

You are given a singly linked list of length n. The goal is to split it into two halves: if n is odd, the extra node belongs to the first half.

Examples:

n = 6: 1 → 2 → 3 → 4 → 5 → 6 becomes (first: 1 → 2 → 3, second: 4 → 5 → 6)

n = 5: 1 → 2 → 3 → 4 → 5 becomes (first: 1 → 2 → 3, second: 4 → 5)

n = 2: 1 → 2 becomes (first: 1, second: 2)

n = 1: 1 becomes (first: 1, second: NULL)

Someone has tried the following algorithm to solve this problem:

```
if head == NULL or head.next == NULL:
    return (head, NULL)

slow = head
fast = head.next
while fast != NULL and fast.next != NULL:
    slow = slow.next
    fast = fast.next.next

second = slow.next
slow.next = NULL
return (head, second)
```

Claim: This algorithm always correctly splits the list into two parts of sizes satisfying the requirements.

Select the option containing the correct response with best reasoning regarding the claim:

a. False, because initializing fast as head.next ensures the extra node goes to the second half when n is odd.

b. False, because in the loop slow advances once while fast advances twice, which causes the split position to be misplaced.

c. True, because initializing fast as head ensures that for odd n, slow stops before the midpoint.

d. True, because with fast = head.next, for odd n the slow pointer stops at the last node of the first half; then second = slow.next and slow.next = NULL correctly split the list into two parts, covering all nodes without overlap. **(Ans)**

e. False, because when n = 2, the loop is skipped and second becomes NULL, showing the algorithm fails for small even lists.

Explanation:

a: Incorrect. When n is odd, the extra node actually goes to the first half, not the second.

b: Incorrect. The fact that slow moves one step and fast moves two steps is exactly what ensures the split point is correct. It doesn't misplace the cut.

c: Incorrect. The given algorithm initializes fast = head.next and not head, that is it refers to the second node not the first. If we set fast = head, then fast may move more steps, hence slow may wrongly stop AFTER the midpoint, instead of before as claimed.

d: Correct. The algorithm is designed so that when n is odd, slow stops at the last node of the first half. Breaking the link at slow.next gives two valid halves with sizes ⌈n/2⌉ and ⌊n/2⌋, and all nodes are preserved without overlap.

e: Incorrect. For n = 2, the loop is skipped, but second = slow.next is the second node, not NULL. Both halves are returned correctly.

Q11 First Non-Repeating Element [7 marks]

Problem statement: Given an integer array `A` of length `n`, return the index `i` of the first element whose value appears exactly once in the array. If no such index exists, return `-1`.

Assume 0-based indexing.

Example:

Input: `A = [2, 3, 5, 3, 2, 4, 5]`

Output: `i = 5 # A[5] = 4 appears exactly once; all earlier elements repeat.`

Correct algorithms with time:

O(n) average or better - 7 marks

O(n log n) - 5 marks

$O(n^2)$ - 4 marks

Answer:

1. Let C be an empty hash map from integer -> integer. // value -> count
2. Let i = 0.
3. while i < n
4. Let x = A[i].
5. if C does not contain key x

6. Set C[x] = 0.

7. Set C[x] = C[x] + 1.

8. i = i + 1

9. Let j = 0.

10. while j < n

11. if C[A[j]] = 1

12. Output j.

13. return

14. j = j + 1

15. Output −1

16. end

Explanation:

We need the earliest index whose value occurs once. A two-pass hash map solution works in linear time: first pass counts occurrences for each value using a map (value → count). The second pass scans A from left to right and returns the first index j with count 1. Hash lookups/updates are average case O(1), so the total runtime is average O(n) with O(n) extra space. In the example, counts are {2:2, 3:2, 5:2, 4:1}; scanning again, the first with count 1 is A[5] = 4, so we return 5.