B

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 {
                return arr[i] + func(arr, i + 2);
        }
}
```

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

Explanation:

The function func uses recursion to process the array, but the recursive call only advances by 2 each time (i.e., i + 2). This means it makes approximately n/2 recursive calls, where each call performs a constant-time operation (accessing arr[i]). Therefore, the time complexity is linear with respect to n, resulting O(n).

Q2 [3 marks]

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

```
int total = 0;
for (int i = 1; i <= n; i++) {
        for (int j = 0; j < n; j++) {
                int t = 1;
                while (t <= j) {
                        total++;
                        t *= 3;
                }
        }
}
```

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

Explanation:
The outer loop runs up to n. For each i, the middle loop runs n times, and each time the while loop takes about log j steps since t grows by powers of 3. So the cost for one i is about n * log n. Adding this up from 1 to n gives a total on the order of n^2 log n. Thus the time complexity is $O(n^2 \log n)$.

Q3 [3 marks]

Consider an empty singly linked list with only a head pointer (no tail pointer). Suppose we perform a sequence of n insertions, where each insertion adds a new element to the end of the list (thus the new element ends up as the last element).

What is the total time complexity of these n insertions?

a. O(n) **b.** O(n logn) **c.** O(n^2) **(Ans)** **d.** O(1) **e.** O(n^3)

Explanation:
Since there is no tail pointer, inserting at the end requires traversing the entire list each time to find the last node. The first insertion requires 0 steps to traverse (just one node), the second requires 1 step, the third requires 2 steps, ..., the n-th insertion requires (n−1) steps in traversal.

Total traversal steps: $1 + 2 + 3 + ... + (n-1) \approx n^2/2$, which is O($n^2$). Each insertion takes O(k) time, where k is the current length of the list, so the overall time is quadratic. Even though the data is inserted once per operation, the repeated traversal makes the total cost O(n^2).

Q4 [3 marks] A queue is initially empty. After a sequence of enqueue and dequeue operations, which of the following sequences of operations results in element 25 being at the front of the queue in the final state?

I. Enqueue 10 → Enqueue 15 → Dequeue → Enqueue 20 → Enqueue 25 → Dequeue → Enqueue 30 → Dequeue → Dequeue

II. Enqueue 5 → Enqueue 10 → Enqueue 15 → Dequeue → Dequeue → Enqueue 20 → Enqueue 25 → Enqueue 15 → Dequeue → Dequeue

III. Enqueue 15 → Enqueue 15 → Enqueue 20 → Dequeue → Enqueue 22 → Dequeue → Dequeue → Enqueue 25 → Dequeue

IV. Enqueue 12 → Enqueue 15 → Enqueue 17 → Dequeue → Dequeue → Enqueue 20 → Enqueue 25 → Enqueue 30 → Dequeue

V. Enqueue 5 → Enqueue 15 → Dequeue → Enqueue 19 → Enqueue 20 → Dequeue → Dequeue → Enqueue 25 → Dequeue → Dequeue

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

Explanation:
I: [10]→ [10,15]→ [15]→ [15,20]→ [15,20,25]→ [20,25]→ [20,25,30]→ [25,30]→ [30]

II: [5]→ [5,10]→ [5,10,15]→ [10,15]→ [15]→ [15,20]→ [15,20,25]→ [15,20,25,15]→ [20,25,15]→ [25,15]

III: [15] → [15,15]→ [15,15,20]→[15,20]→ [15,20,22]→ [20,22]→ [22]→ [22,25]→ [25]

IV: [12]→ [12,15]→ [12,15,17]→ [15,17]→ [17]→ [17,20]→ [17,20,25]→ [17,20,25,30]→ [20,25,30]

V: [5]→ [5,15]→ [15]→ [15,19]→ [15,19,20]→ [19,20]→ [20]→ [20,25]→ [25]→ []

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

You are developing a system to organize a list of students and their grades in a school database. Each student has a name, student ID, and a grade for a specific subject. The system needs to sort the students based on their grades in ascending order (lowest to highest). The sorting algorithm you choose uses the following logic:

1. Find the minimum grade in the list.
2. Swap the minimum grade with the first unsorted element.
3. Repeat the process with the remaining unsorted elements.

Q5 [3 marks]
What is the time complexity of the sorting algorithm used to sort the students based on their grades?

a. O(1) **b.** O(n) **c.** O(n^2) **(Ans)** **d.** O(n log n) **e.** O(n^2 log n) **f.** None of the above

Explanation:
The algorithm described follows the logic of **selection sort**. In selection sort, for each element in the list, the algorithm scans the remaining unsorted part of the list to find the minimum element, which takes O(n) time. This process is repeated for each element, and since there are nnn elements, the total time complexity is O(n^2), where n is the number of students in the database.

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 student list itself.)

a. O(1) **(Ans)** **b.** O(n) **c.** O(n^2) **d.** O(n log n) **e.** O(n^2 log n) **f.** None of the above

Explanation:
The selection sort algorithm works **in-place**, meaning that it sorts the list without requiring any additional space besides a constant amount of space for temporary variables (such as the one used to swap elements). As a result, the space complexity is O(1), regardless of the number of students in the database. This makes selection sort an efficient algorithm in terms of space usage.

Q7 [3 marks]

A hash table with 13 slots uses chaining for collision resolution. The hash function is h(k) = k mod n. The table starts off empty, then 26 keys are inserted, where each key is chosen at random from a set of 10n possible keys. If the hash function uniformly distributes keys across the 13 slots, what is the expected number of keys in the chain at slot 0 after all m = 26 keys are inserted?

a. 1 **b.** 2**(Ans)** **c.** 3 **d.** 4 **e.** None of the above

Explanation:
The hash table has n = 13 slots, and m = 26 keys are inserted. The hash function is h(k) = k mod 13, and keys are chosen uniformly at random from a set of 10n = 130 possible keys. With chaining, each key hashes to one of the 13 slots, and we need the expected number of keys in the chain at slot 0. Since the hash function distributes keys uniformly, the probability that a key hashes to slot 0 is 1/13. The number of keys hashing to slot 0 follows a binomial distribution: Binomial(26, 1/13). The expected number of keys in slot 0 is: E[number of keys in slot 0] = m * 1/n = 26 * 1/13 = 2.
Thus, the expected number of keys in the chain at slot 0 is 2.

Q8 [3 marks]

In a Max-Heap-based Priority Queue, where is a new element inserted before any restructuring occurs?

a. At the root of the heap **b.** At a random position in the heap **c.** As the left child of the root

d. At the next available position at the bottom level of the heap**(Ans)** **e.** None of the above

Explanation:

When inserting a new element into a Max-Heap-based Priority Queue, the element is first placed at the next available position at the bottom level of the heap. After insertion, a "bubble-up" operation is performed to restore the heap property by comparing the new element with its parent and swapping if necessary.

Q9 [5 marks]

Consider an array of length n that is already sorted in a strictly increasing order: A = [1, 2, …, n-1, n].

Compare 2 sorting algorithms to sort A in **nondecreasing** 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.

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

The optimized version finds no swaps in the first pass and stops, which is $\Theta(n)$. The standard version still executes all n−1 passes, which is $\Theta(n^2)$, so they are not the same.

Q10 [4 marks]

You are given the head pointer of a singly linked list with n nodes. Reorder the list in place so that all nodes at odd **1-based** indices (1st, 3rd, 5th, …) appear first, followed by all nodes at even **1-based** indices (2nd, 4th, 6th, …). Preserve the relative order within the odd group and within the even group.

Examples:

n = 6: 1 → 2 → 3 → 4 → 5 → 6 becomes 1 → 3 → 5 → 2 → 4 → 6

n = 5: 1 → 2 → 3 → 4 → 5 becomes 1 → 3 → 5 → 2 → 4

Someone has tried the following algorithm to solve this problem:

```
if head == NULL or head.next == NULL:
    return head
odd = head
even = head.next
evenHead = even
while even != NULL and even.next != NULL:
    odd.next = even.next
    odd = odd.next
    even.next = odd.next
    even = even.next
odd.next = evenHead
return head
```

Claim: This algorithm always correctly reorders the list into all odd-indexed nodes followed by all even-indexed nodes, preserving their relative order.

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

a. True, because manipulating the next references within the while loop makes the odd-indexed nodes keep their original order while the even-indexed nodes are reversed, and then appending evenHead yields the required order.

b. False, because starting with even = head.next causes the last odd node to be dropped when n is odd.

c. True, because the loop hops odd → odd and even → even, building two chains (odds, evens) following the original relative order within the chain; finally odd.next = evenHead concatenates them into exactly "all odds then all evens," with no loss or duplication, including boundary cases. **(Ans)**

d. False, because for n = 2 (e.g., 1 → 2) the loop executes once and the list becomes 1 → 2 → 2.

e. False, because we can omit evenHead and just set odd.next = even at the end without affecting correctness.

Explanation:

a: Incorrect (reason is wrong). The algorithm does not reverse the even-indexed nodes; it preserves their original order while separating odds and evens. Although the claim ("algorithm is correct") is true, the explanation is wrong, so the option is marked incorrect.

b: Incorrect. When n is odd, the last odd node is retained via execution of odd.next = even.next in the last iteration over the 3rd last and 2nd last node; even is NULL but odd is not, so no node is dropped.

c: Correct. The algorithm forms two stable subsequences (odds and evens) and then joins them using the saved evenHead, yielding the required arrangement without losing or duplicating nodes.

d: Incorrect. For n = 2 the loop does not run; the list remains 1 → 2, which already satisfies "odds then evens."

e: Incorrect. Without evenHead you lose the reference to the start of the even list at the end; even typically points at the tail of the even chain, so you cannot correctly stitch all even nodes after the odds.

Q11 First Non-Repeating Element [7 marks]

Problem statement: Given a string S[1…n] consisting of lowercase letters, repeatedly delete any adjacent equal pair until no such pair remains. Return the final string, or a character array containing only the final string.

Example 1: S = "abbaca" → delete "bb" → "aaca" → delete "aa" → "ca".

Example 2:

S = "abddbaccca" → delete "dd" → "abbaccca" → delete "bb" → "aaccca" → delete "aa" → "ccca" → delete "cc" → "ca"

Correct algorithms with time:

O(n) average or better - 7 marks

O(n log n) - 5 marks

$O(n^2)$ - 4 marks

Hint: Use stack to optimize the run.

Answer:

```
1.  Let Stack be empty.               // will keep characters with adjacent duplicates removed
2.  Let i = 1.
3.  while i <= n
4.      Let ch = S[i].
5.      if Stack is not empty AND top(Stack) = ch
6.          pop Stack.                // remove the adjacent duplicate pair
7.      else
8.          push ch onto Stack.       // keep this character
9.      i = i + 1
10. end
11. Let m = size(Stack).
12. Let R be an array of length m of characters.
13. Let j = m.
14. while Stack is not empty
15.     R[j] = top(Stack).
16.     pop Stack.
17.     j = j - 1
18. end
19. Output the string formed by R[1..m]. return.
```

Explanation:

We scan S left to right and use a stack to cancel adjacent duplicates on the fly. If the current character equals the stack top, we pop to delete the pair; otherwise, we push it. Each character is pushed at most once and popped at most once, so the algorithm runs in O(n) time with O(n) space in the worst case. After the scan, the stack contains the final characters in order from bottom to top; we pop them into an array from right to left to produce the final string (e.g., for "abbaca", the stack evolution yields "c","a" → output "ca").