National University of Singapore

School of Computing

IT5003 - Data Structures and Algorithms

(S1 AY 2026/27; Sat, 21 November 2026)

Time Allowed: 2 hours

INSTRUCTIONS TO CANDIDATES:

1. Do **NOT** open this assessment paper until you are told to do so.
2. This assessment paper comprises FOURTEEN (14) printed pages, including this page.
3. This is an **Open Book Assessment**.
   You cannot use any electronic device except one calculator.
4. You can use either pen or pencil. Just make sure that you write **legibly**!
5. Important tips: Pace yourself! Do **not** spend too much time on one (hard) question.
   Read all the questions first! Some (sub-)questions might be easier than they appear.
   Question B.1 and B.2 have Generative AI answers.
   Analyze the Generative AI answers; improve them if you can.
6. For all questions in Section B,
   you can use **pseudo-code** in your answer but beware of penalty marks for **ambiguous answers**.
   You can use **standard, non-modified** classic algorithm in your answer by just mentioning its name, e.g. run Inorder Traversal on BST T, DFS on graph G, Dijkstra's on graph G', etc.
7. For this paper, we assume that Quick Sort algorithm runs in $O(n \log n)$ and hash table insert/search/remove operations are $O(1)$.
8. The total marks is 75. All the best :)

A MCQs ($5 \times 3 = 15$ marks)

Select the **best unique** answer for each question.
Each correct answer is worth 3 marks.

1. A new distinct key is inserted into a valid AVL Tree. In class, we have learned that this new distinct key will appear as a leaf vertex l. Suppose this insertion cause unbalanceness at more than one vertex. Now let z be the first unbalanced ancestor encountered while moving upward from leaf vertex l. The appropriate single (L/R) or double (LR/RL) rotation is applied to rebalance this subtree rooted at z.

   Which statement explains why no further rotations are needed above the repaired subtree?

   a). The repaired subtree has the same height as it had before the insertion

   b). An insertion can make at most one vertex unbalanced

   c). The rotations set every higher ancestor's balance factor to 0

   d). Rotations leave the height of every individual vertex unchanged

   e). Preserving the inorder sequence guarantees that every higher ancestor is balanced

2. Read Section B.3 in page 5 about functional graphs and the proposed graph data structure, which stores one successor vertex for each vertex. For a functional graph fG with V vertices and $E = V$ edges, what is the *tightest* worst-case asymptotic space complexity of this representation?

   Notice the keyword 'tightest', i.e., if the best answer is x but you choose an option that is worse than x (which is still true in Big O notation, but not the tightest), you will be marked as wrong. Similarly if you choose an option that is better than x (impossible), you will be marked as wrong.

   a). $O(V^2)$

   b). $O(V \log V)$

   c). $O(V)$

   d). $O(\log V)$

   e). $O(1)$

3. Consider a directed graph whose vertices represent five cities: Singapore, Bangkok, Hanoi, Tokyo, and Seoul. The graph has exactly the following four directed edges:

   - Singapore → Hanoi
   - Bangkok → Hanoi
   - Hanoi → Tokyo
   - Tokyo → Seoul

   Which of the following sequences is a valid topological ordering of the graph?

   a). Singapore, Hanoi, Bangkok, Tokyo, Seoul
   b). Hanoi, Singapore, Bangkok, Tokyo, Seoul
   c). Singapore, Bangkok, Tokyo, Hanoi, Seoul
   d). Singapore, Bangkok, Hanoi, Tokyo, Seoul
   e). Seoul, Singapore, Bangkok, Hanoi, Tokyo

4. A directed graph has strictly positive edge weights. Which transformation is guaranteed to preserve exactly the same shortest paths from the source, including ties, for every such graph?

   a). Add 7 to every edge weight
   b). Square every edge weight
   c). Multiply every edge weight by 7
   d). Replace every edge weight with 1.
   e). Replace every edge weight w with $1/w$.

5. A directed graph has strictly positive edge weights, and every vertex is reachable from source vertex s. Let $d[i]$ denote the shortest-path distance from s to vertex i *before* any modification. Now, a new directed edge $u \to v$, with weight $w > 0$, is added. Which condition is necessary and sufficient for at least one shortest-path distance from source vertex s to decrease?

   a). $w < d[v]$
   b). $d[u] < d[v]$
   c). $d[v] + w < d[u]$
   d). $d[u] + w < d[v]$
   e). $w < d[u] + d[v]$

B Application Questions ($4 \times 15 = 60$ marks)

B.1 Unique Frequency (15 marks)

Given an integer array `nums` with n ($n \geq 1$) positive integers not larger than 10^{12}, your task is to return an integer denoting the first integer in `nums` whose frequency is unique. That is, no other integer appears the same number of times in `nums`. If there is no such integer, return `-1`.

Example 1: If `nums = [6, 7, 4, 4]`, then we return `4` because `6` appears once, `7` also appears once (thus this frequency is not unique), and `4` is the first (and only) integer that appears twice.

Example 2: If `nums = [7, 8, 8, 8, 9, 9]`, then we return `7` because `7` appears once, `8` appears 3 times, and `9` appears 2 times. All distinct integers have unique frequencies, and `7` appears first.

Example 3: If `nums = [6, 7]`, then we return `-1` because both `6` and `7` appear once (not unique).

Solve this problem in Python (or pseudo-code).
To break ties among multiple correct solutions, prioritize in order:

1. Tightest worst-case time complexity.
2. Smallest auxiliary space complexity.

B.2 Range Queries (15 marks)

You are given the `root` of an Adelson-Velskii Landis (AVL) Tree consisting of n ($n \geq 1$) distinct positive integers not larger than 10^9 and an array `queries` of size m ($m \geq 1$) consisting of positive integers, also not larger than 10^9. For this task, $n > m$, i.e., there are (much) more integers in the AVL Tree than the number of queries. Your task is to compute and return an integer array `ans` of size m where `ans[i] = max_i - min_i`, where:

- `min_i` is the largest value in AVL Tree $\leq$ `queries[i]`.
  If such a value does not exist, set `min_i = queries[i]`.
- `max_i` is the smallest value in AVL Tree $\geq$ `queries[i]`.
  If such a value does not exist, set `max_i = queries[i]`.

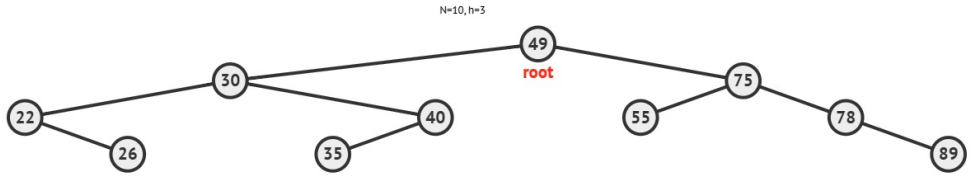


Figure 1: Example AVL Tree for B.2; check `queries` in the elaboration below

Example: Refer to the AVL Tree shown in Figure 1 with $n = 10$ integers.
If `queries = [77, 90, 20]` with $m = 3$ queries, then we return `[3, 1, 2]` because:

- For `queries[0] = 77`, `max_i = 78` and `min_i = 75`, thus `ans[0] = 78-75 = 3`,
- For `queries[1] = 90`, `max_i = 90` (it does not exist) and `min_i = 89`; `ans[1] = 90-89 = 1`,
- For `queries[2] = 20`, `max_i = 22` and `min_i = 20` (it does not exist); `ans[2] = 22-20 = 2`.

Solve this problem in Python (or pseudo-code).
To break ties among multiple correct solutions, prioritize in order:

1. Tightest worst-case time complexity.
2. Smallest auxiliary space complexity.

B.3 Functional Graph (15* marks)

In class, we learned a few special graphs, e.g., Tree, Complete Graph, Bipartite Graph, and Directed Acyclic Graph. Now let's learn one more on the spot: Functional Graph fG: A directed graph with n vertices (numbered with 0-based indexing, i.e., $[0..n-1]$), where **each vertex has exactly one outgoing edge** to *another* vertex in fG.

In class, we learned a few Graph Data Structures to store our graphs, e.g., Adjacency Matrix (AM), Adjacency List (AL), and Edge List (EL). Now let's learn one more on the spot, that is specially designed for functional graph fG. We can store the (only) edge of each vertex of fG in a **Direct Addressing Table (DAT)** called `f` of size n, where `f[u] = v` describes the only edge $u \rightarrow v$ that goes out from vertex u ($n = $ `len(f)`$, n \geq 2$; $\forall u \in [0..n-1] : 0 \leq$ `f`$[u] < n$ and `f`$[u] \neq u$).

One more definition: The **score** of a vertex u is the sum of the vertex numbers of all the vertices that have an edge pointing into vertex u. A vertex with no incoming edge has score 0.

Now your task is to return the vertex number with the **highest** score. However, if multiple vertices have the same score, return the one with the **largest** vertex number.

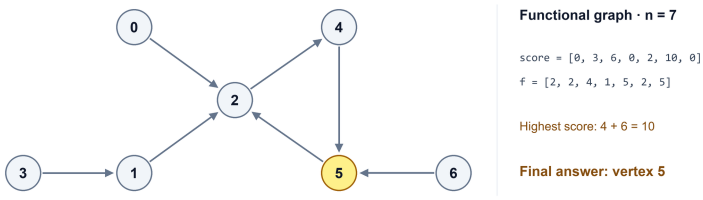


Figure 2: Example fG_1 with $n = 7$ and `f = [2, 2, 4, 1, 5, 2, 5]`

Example 1: For fG_1 shown in Figure 2, we return 5.
Explanation: Vertex 5 (score = 4+6 = 10) has the largest score of 10.

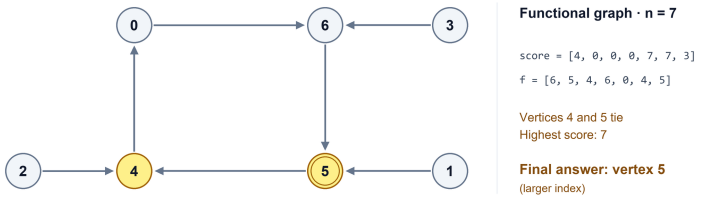


Figure 3: Example fG_2 with $n = 7$ and `f = [6, 5, 4, 6, 0, 4, 5]`

Example 2: For fG_2 shown in Figure 3, we return 5.
Explanation: Both vertex 4 (score = 2+5 = 7) and vertex 5 (score = 6+1 = 7) have the same largest score of 7, but vertex 5 is the larger vertex number.

Solve this problem in Python (or pseudo-code).
To break ties among multiple correct solutions, prioritize in order:

1. Tightest worst-case time complexity.
2. Smallest auxiliary space complexity.

B.4 Assembly Line Scheduling (15* marks)

A factory produces n ($n \geq 1$) different components, labeled from 1 to n, as part of a larger manufacturing project. You are given a 2D array `dependencies` of size $m \times 2$ ($0 \leq m \leq n \cdot (n-1)$), where `dependencies[i] = [A_i, B_i]` indicates that component `A_i` must be fully manufactured before component `B_i` can begin production, where $1 \leq$ `A_i`, `B_i` $\leq n$ and `A_i` $\neq$ `B_i`.

Each production cycle, the factory can manufacture any number of components simultaneously (you can imagine that it is a very resourceful factory), as long as every component being manufactured in that cycle has already had all of its required dependencies completed in a previous production cycle.

Return the minimum number of production cycles required to manufacture all n components. If it is impossible to manufacture all components (for example, due to a circular dependency), return -1.

Example 1:
Input: $n = 3$, `dependencies = [[1, 3], [2, 3]]`
Output: 2
Explanation: In cycle 1, components 1 and 2 are manufactured.
In cycle 2, component 3 is manufactured (since both its dependencies are complete).

Example 2:
Input: $n = 4$, `dependencies = [[1, 3], [2, 3], [3, 4]]`
Output: 3
Explanation: As above, then in cycle 3, component 4 is manufactured.

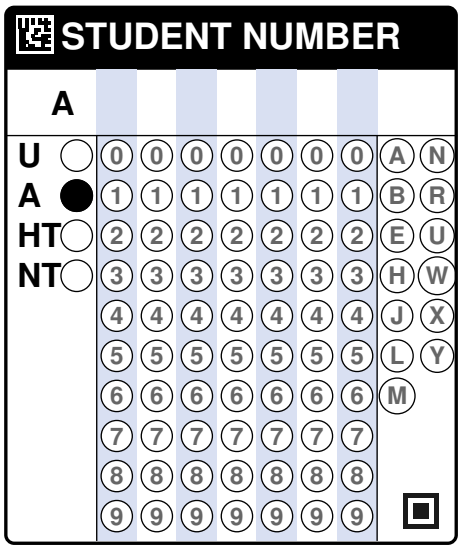
Example 3:
Input: $n = 3$, `dependencies = [[1, 2], [2, 3], [3, 1]]`
Output: -1
Explanation: No valid order exists due to the circular dependency among components 1, 2, and 3.

Solve this problem in Python (or pseudo-code).
To break ties among multiple correct solutions, prioritize in order:

1. Tightest worst-case time complexity.
2. Smallest auxiliary space complexity.

The Answer Sheet for S1 AY2026/27

Write your Student Number in the box below using **pencil**. **Do NOT write your name**.



This portion is for examiner's use only

Section	Maximum Marks	Your Marks	Grading Remarks
A	15		
B	60		
Total	75		

Shade your MCQ answers for Section A below.

No	1	2	3	4	5
A	◯	◯	◯	◯	◯
B	◯	◯	◯	◯	◯
C	◯	◯	◯	◯	◯
D	◯	◯	◯	◯	◯
E	◯	◯	◯	◯	◯

Box B.1. Unique Frequency

```
class SolutionTwin:
    def firstUniqueFreq(self, nums: list[int]) -> int:
        n = len(nums)
        uniqueNums = []
        frequencies = []

        # Find all unique elements and compute frequencies
        for i in range(n):
            if nums[i] not in uniqueNums:
                uniqueNums.append(nums[i])
                count = 0
                for j in range(n):
                    if nums[j] == nums[i]:
                        count += 1
                frequencies.append(count)

        # Mark which frequencies are strictly unique
        isFreqUnique = [False] * len(frequencies)
        for i in range(len(frequencies)):
            unique = True
            for j in range(len(frequencies)):
                if i != j and frequencies[i] == frequencies[j]:
                    unique = False
                    break
            isFreqUnique[i] = unique

        # Scan original array to find the first valid element
        for i in range(n):
            idx = uniqueNums.index(nums[i])
            if isFreqUnique[idx]:
                return nums[i]

        return -1
```

My analysis of SolutionTwin is ($5 \times 1 = 5$ marks):

Ignoring time complexity, it gives correct output for all valid inputs: ◯ YES | ◯ NO, I will elaborate

Ignoring correctness, it runs in: ◯ $O(n^2)$ | ◯ $O(n \log n)$ | ◯ $O(n)$ | ◯ $O(\log n)$ | ◯ Other, $O($_____$)$

The **auxiliary** space that it uses is: ◯ $O(n^2)$ | ◯ $O(n \log n)$ | ◯ $O(n)$ | ◯ Other, $O($_____$)$

It can crash on at least one valid corner case input: ◯ YES, I will elaborate | ◯ NO

I can improve upon the Generated AI code: ◯ YES, I will elaborate | ◯ NO, it is the best possible

Elaboration of my analysis above (10 marks):

Only if you need to improve upon the Generated AI code:
My faster+correct code runs in: ◯ $O(n \log n)$ | ◯ $O(n)$ | $O(\log n)$ | ◯ Other, $O($_____$)$, and the **auxiliary** space that it uses is: ◯ $O(n^2)$ | ◯ $O(n \log n)$ | ◯ $O(n)$ | ◯ Other, $O($_____$)$.

Box B.2. Range Queries

```
from bisect import bisect_left, bisect_right
# bisect_left: Returns the index before any existing copies of the target value.
# If the value already exists in the list, it targets the first occurrence.
# bisect_right: Returns the index after any existing copies of the target value.
# If the value already exists in the list, it targets the last occurrence + 1.

# All other AVLTree details (e.g., rotations) are omitted.
class AVLNode:
    def __init__(self, val: int):
        self.val = val
        self.height = 0
        self.left = None
        self.right = None

class SolutionShannon:
    def ans(self, root: AVLNode, queries: list[int]) -> list[int]:
        values = set()
        self.inorder(root, values)
        sortedValues = sorted(values)

        m = len(queries)
        ans = [0] * m
        for i, q in enumerate(queries):
            floorIdx = bisect_right(sortedValues, q) - 1
            ceilIdx = bisect_left(sortedValues, q)

            floor = sortedValues[floorIdx] if floorIdx >= 0 else None
            ceil = (
                sortedValues[ceilIdx]
                if ceilIdx < len(sortedValues)
                else None
            )

            minI = q if floor is None else floor
            maxI = q if ceil is None else ceil
            ans[i] = maxI - minI

        return ans

    def inorder(self, node: AVLNode, values: set[int]) -> None:
        if node is None:
            return
        self.inorder(node.left, values)
        values.add(node.val)
        self.inorder(node.right, values)
```

My analysis of SolutionShannon is ($5 \times 1 = 5$ marks):

Ignoring time complexity, it gives correct output for all valid inputs: ◯ YES | ◯ NO, I will elaborate

Ignoring correctness, it runs in: ◯ $O(n^2)$ | ◯ $O(n \log n)$ | ◯ $O(n)$ | ◯ $O(\log n)$ | ◯ Other, $O($_____$)$

The **auxiliary** space that it uses is: ◯ $O(n \log n)$ | ◯ $O(n)$ | ◯ $O(\log n)$ | ◯ Other, $O($_____$)$

It can crash on at least one valid corner case input: ◯ YES, I will elaborate | ◯ NO

I can improve upon the Generated AI code: ◯ YES, I will elaborate | ◯ NO, it is the best possible

Elaboration of my analysis above (10 marks):

Only if you need to improve upon the Generated AI code:
My faster+correct code runs in: ◯ $O(n \log n)$ | ◯ $O(m \log n)$ | ◯ Other, $O($_____$)$, and
the **auxiliary** space that it uses is: ◯ $O(n)$ | ◯ $O(m)$ | ◯ $O(1)$ | ◯ Other, $O($_____$)$.

Box B.3. Functional Graph (This is a STARRED task without Generative AI assistance)

◯ I give up, just give me the free 2 marks (you must leave the box below totally empty or cross it)
◯ I know how to solve this, I will elaborate below (pseudocode is allowed and encouraged):

My solution runs in $O($_____$)$ and uses $O($_____$)$ **auxiliary** space.

Box B.4. Assembly Line Scheduling (This is a STARRED task without Generative AI assistance)

◯ I give up, just give me the free 2 marks (you must leave the box below totally empty or cross it)
◯ I know how to solve this, I will elaborate below (pseudocode is allowed and encouraged):

My solution runs in $O($_____$)$ and uses $O($_____$)$ **auxiliary** space.

You can use this page as scratch paper for your workings.
This page will NOT be graded.

– END OF PAPER; All the Best –