**Q3) MINIMUM NO OF SEMESTERS**

PROBLEM STATEMENT :

You are given an integer ( n ), representing the number of courses labeled from 1 to ( n ). Additionally, you have an array called relations where relations[i] = [prevCourse\_i, nextCourse\_i] indicates a prerequisite relationship: course prevCourse\_i must be completed before course nextCourse\_i. You are also given an integer ( k ). In one semester, you can enroll in up to ( k ) courses, provided all prerequisites for those courses have been completed in previous semesters. Return the minimum number of semesters required to complete all courses.

APPROACH :

The core idea is to simulate the process of taking courses semester by semester while considering prerequisites and the limit on the number of courses per semester. We aim to maximize the number of courses taken in each semester to minimize the total number of semesters.

The approach used here explicitly maximizes the number of courses that become available in future semesters by selecting the courses that unlock the most subsequent courses. This is done with the help of priority queue. This optimization can reduce the total number of semesters compared to a simple topological sort-based approach.

PRIMARY DATA STRUCTURES USED :

1. **Unordered\_map :**

· unordered\_map<int, int> prerequisite\_count **:** Tracks the number of prerequisites each course has. This is crucial for determining when a course becomes available for enrollment.

· unordered\_map<int, vector<int>> successor\_map **:** Maps each course to the list of courses that depend on it. This structure is used to efficiently update the prerequisite counts and identify new courses that become available after completing a course.

1. **Unordered\_set :**

· unordered\_set<int> available\_courses **:** Stores the set of courses that can be taken in the current semester. This set is dynamically updated as courses are completed and their prerequisites are met.

1. **Vector :**

* vector<vector<int>> relations **: This 2D vector** stores the input pairs of prerequisite relationships between courses.
* vector<int> current\_semester\_courses **:** Holds the courses that will actually be taken in the current semester.

1. **Priority Queue:**

* priority\_queue<pair<int, int>> pq **: This is used to store the courses in the order of no of courses they unlock**.

TIME COMPLEXITY :

-> Initialization of prerequisite\_count and successor\_map :

· These are initialized with a loop over all n courses : **O(n)**.

· Filling these structures involves iterating over the m relations : **O(m)** where m is the number of relations.

-> Main loop :

· The main loop runs as long as there are courses available to take, which could be **O(n)** iterations.

· In each iteration, the algorithm:

* Each insertion operation in a binary heap (priority queue) takes **O(log l)**, where ‘l’ is the number of elements in the heap at that moment.
* The other operations take place in **O(m)**.

-> Total time complexity :

O(n) + O(m) + O(n\*(log n + m))

Thus, **O(n log n + n\*m)** -> worst case

**O(n \* m) -**> best case

SPACE COMPLEXITY :

-> The space used for prerequisite\_count is **O(n)** .

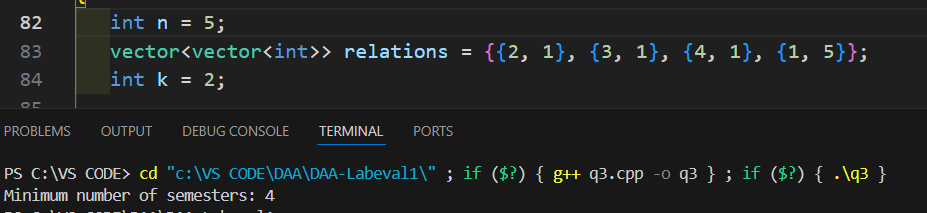
-> The space used for successor\_map is **O(n+m)** .

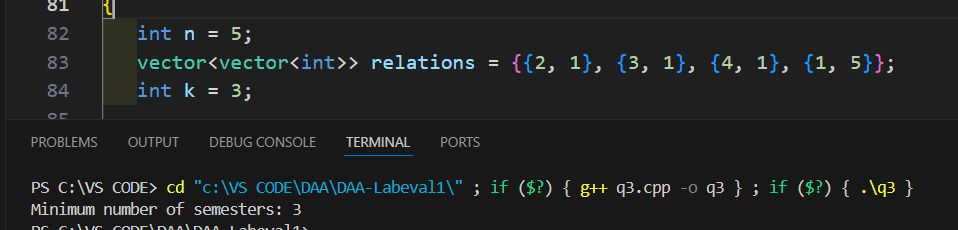
-> available\_courses, current\_courses and next\_semester\_courses : **O(n)** .

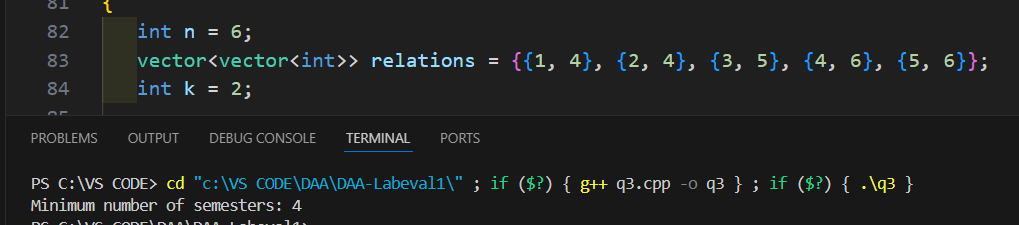
-> pq (priority queue) :  **O(n)**

-> Total : **O(n+m)**

SCREENSHOTS :







COMPARISON :

**Topological Sorting: The courses are sorted based on their dependencies, and in each step, up to k courses are selected based on this topological order. While this approach ensures that prerequisites are respected, it may not be as efficient in minimizing the number of semesters because it does not consider the impact of course selection on future course availability.**

**Our Approach :** The approach used here explicitly maximizes the number of courses that become available in future semesters by selecting the courses that unlock the most subsequent courses. This optimization can reduce the total number of semesters compared to a simple topological sort-based approach, although the time complexity is a bit higher.

OBSERVATION :

When we use a hashmap, both the execution time and the memory usuage is a bit higher than the normal method using topological sorting, but this approach indeed improves the efficiency.

**Q4 a) NUMBER OF ISLANDS**

PROBLEM STATEMENT :

Given an ( m X n ) grid, where each cell contains either a 1 (indicating a valid path) or a 0 (indicating an obstacle), design an iterative algorithm to find the total number of islands present in it.

APPROACH :

Traverse through each cell in the grid using nested loops.

When a 1 is found, increment island\_count and mark all the connected land cells (entire island) as visited ‘0’. This marking ensures that each island is counted exactly once.

we could use recursion to traverse through the entire island to mark it as visited. But since an iterative algorithm is required here, an explicit stack data structure is used to simulate the recursion for converting the recursive depth-first search (DFS) approach into an iterative one.

PRIMARY DATA STRUCTURES USED :

1. **2D Vector:**

* vector<vector<char>> : This stores the input grid and is modified in place to mark visited cells.

1. **Stack :**

* stack<pair<int, int>> stack1 : used to perform the iterative DFS, holding pairs of coordinates (i, j) representing the current cell being explored.

TIME COMPLEXITY :

The time complexity of the algorithm is **O(r \* c)**, where ‘r’ is the number of rows and ‘c’ is the number of columns in the grid. This is because each cell in the grid is processed at most once. The DFS operation, while iterative, still explores each cell's neighbors linearly.

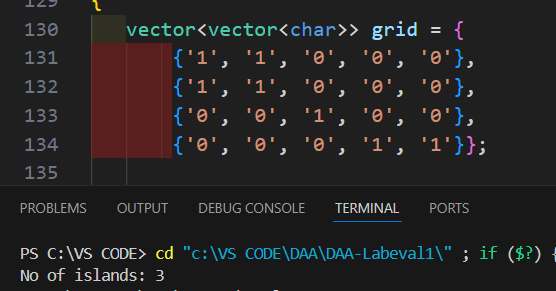
-> **O(r \* c)**

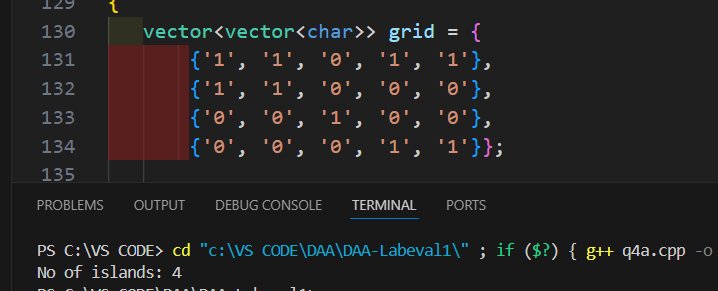
SPACE COMPLEXITY :

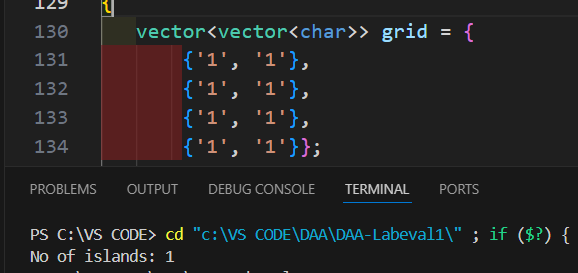
Since the Grid is marked ‘visitied’ is in-place. It doesn’t require additional space, which is O(1). But as we have used explicit stack, In the worst case, the stack can grow to hold all the cells of a single island. If an island covers almost the entire grid, the stack might hold up to O(r \* c) elements.

-> **O(r \* c)**

SCREENSHOTS :







OBSERVATION :

The iterative depth-first search (DFS) approach efficiently identifies and counts the number of islands in the grid. The use of a stack to simulate the recursion ensures that the algorithm remains iterative.

**Q4 b) PATHS & DEAD-ENDS**

PROBLEM STATEMENT :

Given an ( m X n ) grid, where each cell contains either a 1 (indicating a valid path) or a 0 (indicating an obstacle), design an iterative algorithm to check whether the grid contains exactly one path to reach from the source (1,1) to the destination (m,n). Also rewrite the code snippet such that it finds the whether the grid possess exactly ‘*k*’ dead ends.

APPROACH :

The approach starts by implementing a recursive depth-first search (DFS) to explore all possible paths from the source (top-left corner) to the destination (bottom-right corner). The algorithm counts the total number of valid paths and checks for dead ends. A dead end is defined as a cell with only one path leading to it, excluding the start and end cells. The function ‘countPathDeadEnds’ recursively checks each cell, marking it as visited and summing the paths from each direction (up, down, left, right). Dead ends are identified by checking adjacent cells for valid paths and counting the number of such paths.

If exactly one path is found and the number of dead ends matches the desired count 'k', the function confirms the condition. The code ensures each path is explored only once, and dead ends are accurately counted by considering surrounding cells.

PRIMARY DATA STRUCTURES USED :

1. **2D Vector:**

* vector<vector<int>> grid : Stores the input grid.
* vector<vector<bool>> visited\_grid : Keeps track of visited cells to prevent revisiting during DFS.

**ii. Tuple :**

* tuple<int, int> : Used to return both the total path count and the dead-end count from the `findPathDeadend` function.

TIME COMPLEXITY :

The algorithm explores all possible paths from the start to the destination. The worst-case time complexity is O(r\*c), accounting for the four possible directions of movement (up, down, left, right) from each cell.

Finding the number of dead ends takes O(1).

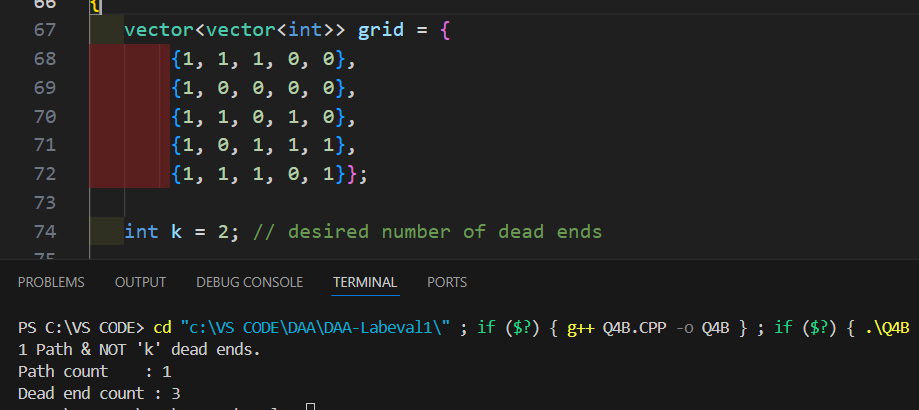
-> **O(r \* c)**

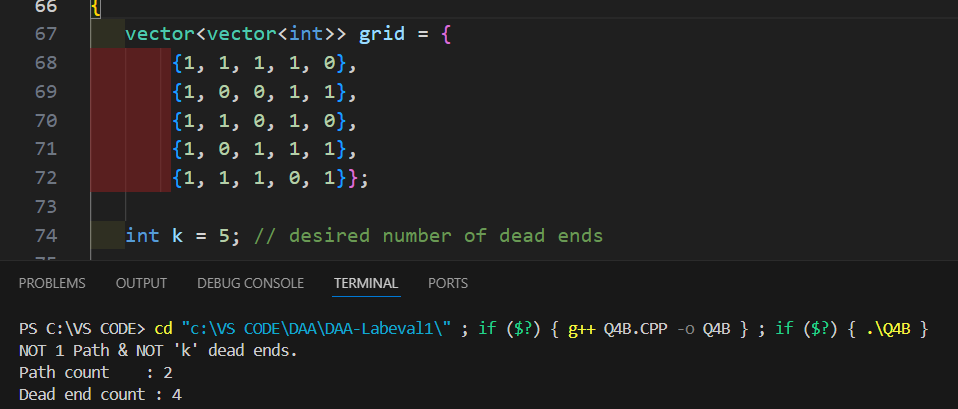
SPACE COMPLEXITY :

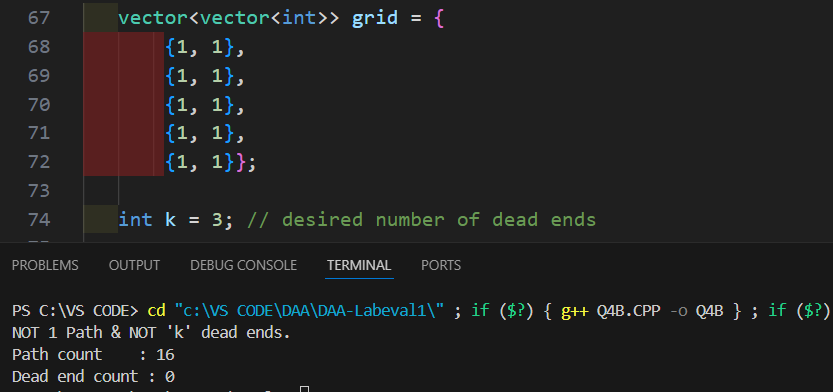
The space complexity is O(r\*c) for the visited grid and the recursion stack in the worst case, where m and n are the grid dimensions.

-> **O(r \* c)**

SCREENSHOTS :







OBSERVATION :

Since the algorithm merges the calculation of paths and dead ends into a single function, it efficiently determines both the number of paths and dead ends.