CS-218-HW3-Basic

Lakhan Kumar Sunilkumar

02 May 2025

Student ID: 862481700

1 A. Water Refill, Codeforce Submission ID: 317779801

Explanation:

The goal of this problem is to find out how long it will take for all students to finish filling up their water containers.

Here is the situation:

- There are n students, each needing a certain amount of water.
- There are m faucets, and each faucet can serve one student at a time.
- Each faucet dispenses exactly one unit of water per second.
- Students stand in a line and go one after another as faucets become available.

So the challenge is to efficiently manage the students and faucets so that we know when the last student finishes. To solve this, we can think of faucets like machines, and students as jobs. We want to schedule the jobs so that the work is balanced and the total finish time is minimized. The best way is to always assign the next student to the faucet that becomes free the earliest. This ensures that no faucet stays idle, and we spread out the work evenly. To manage this efficiently, we use a **min-heap** (also called a priority queue), which always gives us the faucet that will be free first.

The solution steps are:

1. Send the first m students to the m faucets and record when they will finish.
2. For every new student:
   - Pick the faucet that becomes available first.
   - Assign the student to that faucet.
   - Update the faucet's next free time by adding the student's water time.
3. After all students have been processed, the maximum finish time among the faucets is the answer.

Code Structure:

```
Start by reading n (number of students) and m (number of faucets)
Read the water requirement for each student

Initialize a min-heap with the first m students' water times

For each remaining student:
    - Remove the faucet with the smallest finish time from the heap
    - Assign the student to this faucet
    - Add the student's water time to the finish time
    - Push the new finish time back into the heap

After assigning all students,
the answer is the largest value remaining in the heap
```

Time Complexity:

- Initial heap construction with m elements: $O(m)$
- For each of the remaining $n - m$ students:
  - Extract the earliest free faucet: $O(\log m)$
  - Insert the updated free time: $O(\log m)$

Overall Time Complexity: $O(n \log m)$
References

- `https://www.geeksforgeeks.org/job-sequencing-problem/`
- `https://en.wikipedia.org/wiki/List_scheduling`

2 B. Ski, Codeforce Submission ID: 317780723

Explanation:

Well in this problem, we are given a 2D grid representing a ski resort. Each cell in the grid carries a number that represents the elevation at that point. Now, from any given cell, we can move to an adjacent cell either up, down, left, or right, but only when the next cell always has a lower elevation, as one is skiing downhill. Now our objective is to find the longest possible sequence of such downhill moves from any starting cell.

We consider the grid as a graph where each cell is a node, and an acceptable downhill move forms a directed edge from a higher node to a lower one. This graph is a Directed Acyclic Graph (DAG) because you cannot move back to a higher elevation once you've moved to a lower one which is that there are no cycles. For this problem, I have used Depth-First Search (DFS) with memoization, which is also known as top-down dynamic programming where the idea is:

- For each cell, explore all valid downhill moves using DFS.
- Store the longest path found from that cell in a separate table.
- At the end, the answer is the maximum value stored in any cell of that table.

This method avoids repeated work and ensures we only compute each cell's longest path once.

Code Structure

```
Initialize a 2D table to store the longest path starting from each cell (all set to 0).

Define the four directions of movement: up, down, left, right.

For every cell in the grid:
    If the longest path from this cell hasn't been computed:
        Use DFS to explore all valid downhill directions.
        For each direction:
            If the next cell is within bounds and has a lower height:
                Recursively compute its longest path.
                Update the current cell's longest path using the result.

Store the maximum of all computed path lengths as the final result.
```

Time Complexity:

- Each cell is visited once during the DFS.
- For each cell, we check up to 4 neighbors.
- Thanks to memoization, no cell's DFS is repeated.

  Overall Time Complexity: $O(r \times c)$, where r is the number of rows and c is the number of columns.

References

- `https://www.geeksforgeeks.org/longest-increasing-path-matrix/`
- `https://www.youtube.com/watch?v=wCc_nd-GiEc`

3 C. Gridiron Gauntlet, Codeforce Submission ID: 317781307

Explanation:

The problem is based on a minigame in Super Mario Party. A player is placed on a $5{\times}5$ grid and needs to avoid getting hit by attackers called Chargin' Chucks. So each attacker will be appearing at a specific time and can attack either a complete row or a complete column, called a track. Now we know that tracks are numbered from 1 to 10, where $1 - 5$ is represented as rows and $6 - 10$ is represented as columns.

At each unit of time:

- Some tracks, which could be either rows or columns are attacked.
- The player can move to an adjacent cell that is up, down, left, right or can stay in place.
- If the player is on a cell being attacked, they are eliminated.

We know that the player can start at any cell on the grid. The challenge is to determine the maximum time the player can survive under these rules.

Possible Constraints:

- At most 10,000 attacks can occur.
- Each attack is specified by a time and a track number.

Plan of attack:

This is a simulation problem that tracks all possible positions the player can occupy safely over time, so we could do:

1. Start with all 25 grid cells marked as "alive" positions at time 0.
2. For each time unit where attacks occur:
   - First, simulate the player's movement: from each current cell, move to all possible adjacent cells or stay.
   - Then, eliminate any cells that fall within the attacked rows or columns for that time.
3. If no alive positions remain after a step, that time is the last moment the player can survive.
4. If the player survives through all the attacks, output the last attack time plus one.

Code Structure (Pseudocode):

```
Initialize a set containing all cells in the 5x5 grid as possible alive positions.
Group all attacks by their time using a dictionary for fast lookup.

For each time step in chronological order:

    While current time is less than the next attack time:
        Move all alive positions to all possible adjacent (or same) cells.
        If no alive cells remain, print the current time and terminate.

    Identify all cells that lie in attacked rows or columns for this time step.
    Remove all such attacked cells from the alive positions.
    Again, move to all possible adjacent (or same) cells from current alive positions.
    If no alive cells remain after the move, print the current attack time and terminate.

If survived through all attacks, print the last attack time + 1.
```

Time Complexity:

- Initial grid state: $O(1)$, since the grid is 5x5.
- For each time unit:
  - Movement from each cell: constant time due to bounded grid (≤ 25 cells).
  - Elimination of attacked cells: $O(1)$ per attack.
- Processing all attacks: $O(n)$, since each event is visited once.

Overall Time Complexity: $O(n)$

References

- https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/
- https://mariowiki.com/Gridiron_Gauntlet