CS-218-HW2-Challenge

Lakhan Kumar Sunilkumar

17 April 2025

Student ID: 862481700

1 A. Move to the ends: Codeforce Submission ID: 317243185

Explanation:

The goal is to minimize moves by preserving the longest consecutive subsequence in left-to-right order. We can move any element to the beginning or end, but elements in this sequence are should not be touched.

So we can perfom these steps:

- Iterate through the array once.
- For each value v, update seq_v[v] = seq_v.get($v-1, 0$) + 1.
- Track the maximum sequence length ('longest') during iteration.

Finally, the minimum number of moves required is ($n -$ longest), where n is the total number of elements in the array.

Code Structure

```
For each value v in array:
    seq_v[v] = seq_v.get(v - 1, 0) + 1
    longest = max(longest, seq_v[v])
Append (n - longest) to results list
```

Time Complexity:

Single pass over array elements: $O(n)$ Hash map updates and lookups per element: $O(1)$ amortized
Overall Time Complexity: $O(n)$ per test case.

References

- `https://www.geeksforgeeks.org/longest-increasing-consecutive-subsequence/`
- `https://takeuforward.org/data-structure/longest-consecutive-sequence-in-an-array/`

2 B. Secret Message: Codeforce Submission ID: 317245888

Explanation:

The plan of attack is to find the lexicographically smallest path from the top-left (1,1) to the bottom-right (N,M) in a given grid, moving only to adjacent cells (up, down, left, right) (DFS path) without revisiting any cell.

So for these steps:

- Start DFS traversal from the (0,0) cell with the initial prefix string.
- Then at each step, check and collect all unvisited neighbors and sort them lexicographically based on the letter it shows.
- Also prune paths early if the current prefix cannot lead us to a better solution compared to the best.
- Upon reaching (n-1, m-1), compare the current path string and update the best result if necessary.

This ensures that all paths are explored in such a way that it always looks ahead for forming the smallest possible string at each decision point.

Code Structure

```
function dfs(x, y, prefix):
    if prefix cannot beat best:
        return
    if (x, y) is bottom-right:
        update best if prefix is smaller
        return
    collect all valid, unvisited neighbors
    sort neighbors lexicographically
    for each neighbor:
        mark neighbor as visited
        call dfs(neighbor.x, neighbor.y, prefix + neighbor.letter)
        unmark neighbor as visited

function smallest_message(grid):
    initialize best as a very large string
    mark (0,0) as visited
    call dfs(0, 0, grid[0][0])
```

Time Complexity:

Exploring paths using DFS (worst case): $O(4^{N\times M})$. Sorting neighbors at each step: $O(1)$ (since at most 4 neighbors) **Overall Time Complexity:** $O(4^{N\times M})$

References

- `https://stackoverflow.com/questions/29376069/lexographically-smallest-path-in-a-nm-grid`
- `https://www.geeksforgeeks.org/print-the-lexicographically-smallest-dfs-of-the-graph-starting-from-`

Side Note:

I Initially tried solving with a heap (priority queue) approach where it always expands the path with the smallest string first. However, it failed on some test cases because heap makes greedy global decisions without considering better local paths.

3 C. Marathon: Codeforce Submission ID: 317253313

Explanation:

The plan of attack is to ensure that every point on the 42,195m marathon track is covered by at least two cameras using the minimum number of placements. Each camera has a pre-determined location and coverage radius.

So first, we place each camera's coverage interval so that it stays within the valid range $[0, 42195]$ meters and ignore any cameras whose intervals do not overlap the track.

Next, we sort all the valid intervals based on their left endpoints to have an efficient left-to-right sweep of the track.

While sweeping from the start of the track, we set and handle two heaps:

- An **active** min-heap that stores the right ends of cameras that have already been chosen.
- A **candidate** max-heap that stores the right ends of cameras that are currently visible but have not yet been selected.

At every critical point particularly at the right end of a camera we perform two key operations:

- Remove any cameras from the active heap whose coverage has already expired (i.e., their right end is before the current position).
- If the number of active cameras covering the position falls below two, we greedily select the next available camera that covers the position and extends coverage as far as possible.

Finally, we move the position only to the next critical event where coverage might drop, that is, when a camera's coverage expires.

This ensures that all points along the track are continuously covered by at least two cameras using the minimal number of placements through a greedy, event-driven strategy.

Code Structure

```
function solve():
    Clamp all cameras to within [0, 42195] meters
    Sort all camera intervals by their starting point

    while pos is not beyond 42195:
        Add all cameras whose start <= pos into 'candidates'
        Remove any cameras from 'active' whose coverage ends before pos

        while less than two cameras cover pos:
            If no candidates can cover pos:
                Return -1 (coverage impossible)
            Else:
                Select the camera with the farthest right end from 'candidates'
                Add its right end to 'active'
                Increment the camera placement count

        Move pos to the smallest right end among active cameras
        Remove all cameras from 'active' that expire exactly at pos

    Return the total number of cameras placed
```

Time Complexity:

- Sorting intervals: $O(n \log n)$
- Heap operations: Each camera is pushed and popped at most once, $O(n \log n)$

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

References

- `https://cs.stackexchange.com/questions/151112/greedy-algorithm-for-postive-interval-covering`
- `https://www.geeksforgeeks.org/minimum-number-of-intervals-to-cover-the-target-interval/`
- `https://dilipkumar.medium.com/sweep-line-algorithm-e1db4796d638`
- `https://www.youtube.com/watch?v=YnIxejYW7cE`

4 D. Defending Hogwarts Express: Codeforce Submission ID: 317256952

Explanation:
The goal of the problem is to determine how many train cars are destroyed after each Death Eater attack. Each train car has:

- A fixed position on a number line.
- A defense value.

Each attack has a center position y, radius b, covering the range $[y - b, y + b]$ and an attack power A.
A train car is destroyed by an attack if:

- Its position lies within the attack range.
- Its defense value is less than or equal to the attack power.

To handle upto 10^5 cars and 10^5 attacks efficiently we should try:

1. Sort all train cars by their positions.
2. Build a Segment Tree on defense values to quickly find the minimum defense in any range.
3. For each attack:
    - Use Binary Search to find indices of cars within the attack range.
    - Repeatedly find and remove the weakest car (with defense $\leq A$) using the segment tree.
    - Count how many cars were destroyed.

Once a train car is destroyed, it is permanently removed and cannot be destroyed again in future attacks.

Code Structure

```
function solve():
    Sort train cars by position

    Build a Segment Tree where each node stores:
        Minimum defense value,
        Index of that defense

    for each attack:
        Calculate the attack range [y - b, y + b]
        Use Binary Search to find the leftmost and rightmost cars in range

        while there is a train car in the range with defense <= attack power:
            Destroy the car (update its defense in Segment Tree to INF)
            Increment the destroyed count for this attack
```

Time Complexity:

- Sorting train cars by position: $O(n \log n)$
- Building Segment Tree: $O(n)$
- Each attack:
  - Binary Search to find range: $O(\log n)$
  - Destroying each affected train car: $O(\log n)$ per car

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

- https://www.geeksforgeeks.org/segment-tree-range-minimum-query/
- https://cp-algorithms.com/data_structures/segment_tree.html
- https://www.youtube.com/watch?v=ciHThtTVNto