CS-218-HW3-Challenge

Lakhan Kumar Sunilkumar

09 May 2025

Student ID: 862481700

1 A. Lazy Piano: Codeforces Submission ID: 318540691

Explanation:

So the problem is about Takahashi, the pianist playing a song on a one directional or linear piano keyboard with k keys, by pressing n notes in a given sequence. So each note specifies which key needs to be pressed, and the pianist is only allowed to use either the left or right hand.

The issue arises when *fatigue* builds up only when a hand moves from one key to another. The fatigue value is calculated as the distance between keys, which is $|x - y|$, where x is the previous position and y is the new key. The goal is to choose which hand presses which key so that the total fatigue at the end of the song is the least.

As we know that the pianist is only allowed to:

- Start with both hands on any keys.
- Move either hand as needed to press the next key.

To solve this, dynamic programming seems to be the most optimal strategy. The intuition is to remember the most efficient hand placements as the music progresses and update them smartly to avoid unnecessary and redundant recalculations.

Plan of Attack:

1. Track the fatigue of one hand staying at its last-used key while the other hand may move to press the next note.
2. At each step, calculate the best hand to use:
    - If the hand used last is moved again, simply add the move cost to all configurations.
    - Otherwise, move the other hand and record the new minimum fatigue.
3. Use an offset variable to track accumulated move costs, minimizing redundant updates to all possible hand positions.

This strategy make sure for the minimum fatigue without the use of brute forcing all possible hand positions for every note.

Code Structure

```
Read number of test cases

for each test case:
    Read the number of notes n and number of keys k
    Read the list of notes to be played (keys to press)

    Initialize dp list of size k+1 to store minimum fatigue
    Initialize offset to 0
    Set the first key pressed as the initial active key

    for each next key in the sequence:
        Store previous key as 'q'

        Compute best fatigue by checking all key positions 'l':
            For each 'l':
                Calculate total cost if the other hand moves to the new key
                Track the minimum such cost

        Add movement cost for moving hand from 'q' to current key

        Update dp[q] with the best configuration-
        -where the other hand was used instead of the same hand

        Set the current key as the last pressed for next iteration

    After processing all notes, the answer is:
        minimum value in dp + the accumulated offset
```

Time Complexity:

- For each of the n notes in a test case:
  - Looping over k possible hand positions: $O(k)$
- So, each test case runs in: $O(n \cdot k)$

Overall Time Complexity: $O(n \cdot k)$ per test case

Reference
https://cp-algorithms.com/dynamic_programming/divide-and-conquer-dp.html

2 B. The Friendship Questionnaire: Codeforces Submission ID: 318543366

Explanation:

So this question is based on a fictional context from *The Big Bang Theory*, where Sheldon evaluates his friends using a questionnaire. Raj fills out the answers, but makes some mistakes. Sheldon's goal is to find the *minimum number of editing mistakes* Raj made while copying the answers.

The mistakes allowed are:

- Substitution: Changing one letter to another (e.g., B $\rightarrow$ D)
- Insertion: Adding an extra letter accidentally
- Deletion: Missing a letter while copying

Our goal is to calculate the minimum number of such operations needed to convert Raj's string into Sheldon's correct answer string. This is a *Levenshtein distance*.

At each distance level d (number of edits so far), we check possible moves:

- Move right (deletion in Raj's string)
- Move down (insertion in Raj's string)
- Move diagonally (match or substitution)

For each move, we try to move forward along the strings as far as characters match. We repeat this until we can reach the end of both strings or reach the edit limit of 100.

Code Structure

```
function compute_min_edits(s, t):
    if s equals t:
        return 0

    Initialize a list to store furthest reachable x-positions
    on diagonals k = x - y for d = 0 edits

    for each edit count d from 1 to 100:
        for each diagonal k in [-d, d]:
            Try to extend from:
                - Deletion: move right from (x-1, y)
                - Insertion: move down from (x, y-1)
                - Substitution: move diagonally from (x-1, y-1)

            Update the furthest x position reachable for that diagonal

            While characters match, move forward (snake)
                If both strings are fully consumed
                    return d

    return 100
```

Time Complexity:

- Each edit level processes up to $2d + 1$ diagonals
- Each diagonal only moves forward where characters match
- Since the number of edits $D \leq 100$, the time complexity is:

Overall Time Complexity:

$$O(D \cdot (n + m)) \text{ where } D \leq 100$$

References

- `https://en.wikipedia.org/wiki/Levenshtein_distance`
- `https://en.wikipedia.org/wiki/Wagner%E2%80%93Fischer_algorithm`

3 C. The Security System: Codeforces Submission ID: 318558124

Explanation:

The problem simulates a security system for Sheldon and Leonard's apartment, which is modeled as an $L \times L$ grid. The goal is to ensure that horizontal sensor beams sent from the left side of each row to the right do not encounter more than k furniture obstacles per row.

Each piece of furniture is represented as a vertical segment spanning from cell (a, b) to (c, d), with the column fixed (i.e., $b = d$). If too many furniture segments block a given row, the sensors won't function properly. In summary, the task is to determine the *minimum number of furniture pieces to remove* so that each row is intersected by at most k vertical segments.

Plan of Attack:

To minimize removals, a **greedy algorithm** is used such that, at every row, removes the furniture segment with the **furthest reach downwards** (i.e., the highest end row). This ensures that a single removal helps future rows as well.

- A max-heap (priority queue) is used to keep track of active furniture pieces.
- Two lists: one tracks when a segment starts, and the other when it ends.
- A boolean array tracks which furniture pieces are removed.

Algorithm Steps:

1. For each row from 1 to L:
    - Add all furniture segments that start at this row to the heap.
    - If the number of currently active (non-removed) segments exceeds k, remove the one with the **largest ending row** to free up as many rows as possible.
    - Remove segments that naturally end at this row from the active count.

This process ensures that at every row, we stay within the limit k, and we remove the minimal number of obstacles.

Code Structure

```
function solve():
    Initialize lists to record start and end of each furniture segment
    Initialize a max-heap to store active furniture (sorted by end row descending)
    Initialize a counter for active furniture on the current row

    For each row r from 1 to L:
        Add all segments starting at row r to the heap
        Increase the active count accordingly

        While active count exceeds the allowed k:
            Remove the segment from heap that ends the latest
            If it's still active and not already removed:
                Mark it as removed
                Decrease active count
                Increment removal counter

        For each segment ending at r:
            If it hasn't been removed, reduce active count
```

Time Complexity:

- Processing each of the n furniture segments: $O(n \log n)$ (due to heap operations)
- Iterating through each of the L rows: $O(L)$
- Insertion and removal from the heap per furniture: $O(\log n)$

Overall Time Complexity: $\mathbf{O((n + L) \log n)}$

References

- https://www.geeksforgeeks.org/minimum-removals-required-to-make-ranges-non-overlapping/
- https://www.geeksforgeeks.org/how-to-implement-interval-scheduling-algorithm-in-python/
- https://blog.heycoach.in/greedy-algorithm-for-interval-scheduling/

4 D. Select Courses: Codeforces Submission ID: 318965708

Explanation:

So there is a list of n courses, where each is with a credit value and could be a prerequisite course. If a course has a prerequisite, then that following prerequisite must be taken in the same quarter. The goal is to select at most m courses to maximize the total credit, without breaking any prerequisite conditions.
So I place the courses in a forest (which is a collection of trees), where each course either has no root or depends on exactly one prerequisite which is it's parent. The main condition is that the course selection must be ancestor-closed, that is selecting a particular course requires selecting all of its prerequisites up to the root.

Plan of Attack:
The problem is approached with **Dynamic Programming in a tree** (tree DP) combined with a **Global Knapsack DP**:

- So each course and its prerequisite relations are represented as a forest using an adjacency list.
- For each tree (rooted at a course with no prerequisite), perform a depth-first search with dynamic programming.
- At each node represented for course, maintain a DP array `dp_u[k]` representing the maximum credit achievable by choosing exactly k courses in that subtree.
- Initialize the DP with selecting 0 and 1 courses (just the root).
- Merge child DP tables into the parent using a knapsack-style convolution, ensuring no descendant is selected without its parent.

Algorithm Steps:

1. Parse input to build the dependency forest.
2. For each root, perform a DFS to compute subtree DP.
3. At each node:
   - Initialize the DP array with base values.
   - For each child, retrieve its DP table and merge it using nested loops.
   - Skip cases where selecting a descendant without its parent would violate constraints.
4. After computing all root subtrees, apply a global knapsack DP:
   - Initialize a global array `dp_all[i]` to store max credit for i total courses.
   - Merge the subtree DP into the global DP across all roots.

Code Structure

```
dfs(u):
    Initialize dp_u = [0, credit[u]]
    For each child v of u:
        Get dp_v via recursive dfs
        Merge dp_v into dp_u via nested loops:
            Skip cases where selecting descendants without the parent
            would violate the prerequisite constraint
    Return the merged dp_u

main():
    Read n, m and build course dependency forest
    For each root, call dfs(root) to compute subtree DP
    Initialize dp_all to store best credit across all subtrees
    For each root's dp_r:
        Merge into dp_all using classic knapsack DP
    Output dp_all[m]
```

Time Complexity:

- Each course is visited once: $O(n)$
- Each DP merge between parent and child can take $O(m^2)$
- Global DP merge for all trees is also $O(m^2)$ per tree

Overall Time Complexity: $\mathbf{O(n \cdot m^2)}$

References

- https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/
- https://codeforces.com/blog/entry/20935
- https://www.geeksforgeeks.org/dp-on-trees-for-competitive-programming/