CS-218-Challenge-Problems

Lakhan Kumar Sunilkumar

10 June 2025

Student ID: 862481700

1 HW4 Challenge: A. Weightlifting: Codeforces Submission ID: 323054436

Explanation:

We are given n barbell plates with weights that are multiples of 0.5, and we are asked to assemble a barbell of exact total weight x, such that both sides of the barbell are balanced with $x/2$ weight each. Each plate can be used at most once, and we are to use the **minimum number of plates** to achieve this configuration.

If it is impossible to reach the target, we must output -1.

Plan of Attack:

The solution includes a smart hybrid approach of optimization followed by a full dynamic programming fallback:

- **Phase 1: Input Parsing and Normalization**
  Convert all decimal weights to integer values by multiplying each by 2. This allows consistent integer processing, making x and $x/2$ directly comparable to the plate values.
- **Phase 2: 1-D Knapsack Shortcut**
  If any plate has weight equal to the target per side ($T = x/2$), it can be used entirely for one side. The problem then reduces to finding a subset of the remaining plates whose total weight is T. This is solved via 1-D dynamic programming. This shortcut drastically reduces search space and computation time.
- **Phase 3: 2-D Dynamic Programming Fallback**
  If no such "heavy plate" exists, we use a 2D DP table to track all possible combinations of left and right sums. The DP tracks the minimum number of plates used to reach each (s_1, s_2) configuration. We stop once we reach (T, T), which means the barbell is balanced and filled.

Algorithm Steps:

1. Parse input: Read the number of plates n, total weight x, and the n decimal plate weights. Normalize by multiplying each weight by 2 and rounding to get integer values.
2. **Greedy Shortcut:**
   - Check if any plate has weight exactly T.
   - If found, assign this plate to one side.

- Solve the remaining side using 1-D knapsack (subset sum DP).
- Prefer higher-indexed plates to match output expectations.

3. **2-D Dynamic Programming:**
   - Use rolling DP layers: previous and current.
   - For each plate, try placing it on the left, on the right, or skipping it.
   - Track the minimum number of plates used for every (s_1, s_2).
   - Use a 3D byte array to record choices for backtracking.
4. **Backtracking:**
   - Starting from (T, T), backtrack using the 'pre' array.
   - Reconstruct the indices of plates placed on the left and right sides.
   - Sort and print the result.

Code Structure

```
solve():
    Parse input values: n, x, and plate weights.
    Normalize all weights to integer half-units.

    Phase 1 - Shortcut Optimization:
        - Search for any plate with weight == T.
        - If found:
            - Use that plate for one side.
            - Solve 1D knapsack subset sum DP on remaining plates.
            - Backtrack to get selected plate indices.
            - Print total plates used, left indices, and right index.
            - Return.

    Phase 2 - 2D Dynamic Programming:
        - Initialize DP_prev for (s1, s2) states.
        - For each plate:
            - Copy DP_prev to DP_cur.
            - Try placing the plate on left and right (if feasible).
            - Update DP_cur and predecessor array accordingly.
        - If (T, T) not reachable, print -1.

    Backtracking:
        - From (T, T), trace back using pre[].
        - Recover which plates went left/right.
        - Print minimum plate count and indices.
```

Time Complexity:

- 1D knapsack shortcut: $O(n \cdot T)$
- 2D DP fallback: $O(n \cdot T^2)$
- Backtracking: $O(n)$

Overall Time Complexity: $\mathbf{O(n \cdot T^2)}$
Space Complexity: $\mathbf{O(T^2)}$ for DP and backtracking arrays.

2 HW4 Challenge: E. Maximum Matching: Codeforces Submission ID: 323054584

Explanation:

We are given a sequence of n ski segments where each segment may or may not be used to perform a trick denoted by a lowercase letter from 'a' to 'z'. Given a regular expression of length m, the goal is to find the **maximum number of tricks** from the input sequence that match the regular expression, possibly by skipping some characters. If no matching routine exists, we print -1.

The regular expression can contain:

- Lowercase letters: matches exactly that character.
- Question mark (?): matches any single character.
- Bracket expressions like [abc]: matches any one of the characters inside.
- Asterisk (*): means the previous item (letter, ?, [], or a grouped expression in ()) may repeat zero or more times.
- Grouped expressions like (ab?)*: parenthesis must be followed by *, and do not contain nested parentheses.

The solution must compute the length of the longest contiguous subsequence of ski tricks matching the regular expression.

Plan of Attack:

This problem requires implementing regex matching via Thompson's construction of an NFA, followed by simulating the automaton using dynamic programming.

- **Step 1: Parse Regular Expression**
  Convert the input pattern into a sequence of tokens, distinguishing types such as characters, wildcards, bracket sets, and grouped expressions.
- **Step 2: Construct NFA using Thompson's Algorithm**
  Build a Non-deterministic Finite Automaton from the token stream. Add ε-transitions for concatenation and Kleene star handling. Record transitions and ε-links.
- **Step 3: Precompute ε-closures**
  For each NFA state, determine the set of states reachable via ε-transitions alone. This is used in both initialization and transition propagation.
- **Step 4: Dynamic Programming Simulation**
  Simulate NFA on the trick string:
  - Track the maximum number of tricks matched at each state.
  - For each character, either skip or match it using transitions.
  - Use ε-closure after each match step to propagate possible states.
- **Step 5: Final Evaluation**
  Among all NFA states reachable from the accept state, return the maximum number of tricks matched. If none, return -1.

Algorithm Steps:

1. Parse the pattern into structured tokens identifying whether an element is starred or part of a group.
2. Build the NFA:
   - Assign new states for each token and handle transitions.
   - Use ε-links for concatenation and zero-or-more (*) repetition.
3. Compute ε-closure of all NFA states using DFS/BFS.
4. Initialize DP state: mark all states in the ε-closure of the start state with trick count 0.
5. For each trick in the input string:
   - Try skipping the character (carry over previous state).
   - Try matching with available transitions.
   - Update DP for states in ε-closure of destination states.
6. After processing all characters, find the best DP value in ε-closure of the accept state.
7. Return that value or -1 if no match was possible.

Code Structure

```
main():
    Read number of test cases.
    For each case:
        - Read n and string of tricks.
        - Read m and regular expression.
        - Parse regex into token list.
        - Build NFA using Thompson's construction.
        - Compute epsilon-closures for all NFA states.
        - Initialize dp array for max trick matches at each state.
        - For each trick in s:
            - Either skip the trick or match it using transitions.
            - Apply epsilon-closure to propagate states.
        - Return max(dp[state]) for states in epsilon-closure of accept.
        - If none are valid, return -1.
```

Time Complexity:

- Parsing and NFA construction: $O(m)$
- ε-closure computation: $O(m^2)$
- DP simulation over string of length n: $O(n \cdot m^2)$

Overall Time Complexity: $\mathbf{O(n \cdot m^2)}$
Space Complexity: $\mathbf{O(n \cdot m)}$ for DP arrays and NFA graph.

References

- https://en.wikipedia.org/wiki/Thompson%27s_construction
- https://www.geeksforgeeks.org/regular-expression-to-nfa/
- https://cp-algorithms.com/string/string-hashing.html