CS-218-HW4-Challenge

Lakhan Kumar Sunilkumar

23 May 2025

Student ID: 862481700

1 B. Palindromic Partition: Codeforces Submission ID: 320834494

Explanation:

We are given multiple test cases, each consisting of a string of digits. The task is to split each string into the maximum number of contiguous substrings such that the sequence of substrings forms a palindrome. In other words, if the parts are $[s_1, s_2, \ldots, s_k]$, then $s_i = s_{k-i+1}$ for all valid i.

For example, the string `"179791"` can be partitioned as `["1", "79", "79", "1"]`, which forms a valid palindromic sequence of 4 parts.

Plan of Attack:

The naive approach of comparing substrings directly becomes inefficient for strings of length up to 10^6. To optimize, we apply a rolling hash technique with two moduli (double hashing) to compare substrings in constant time.

- Use a polynomial rolling hash to compute forward hashes of the input string under two different moduli.
- Use two pointers, one from the left and one from the right, and try to "peel off" matching prefix-suffix substrings.
- For every match found, count 2 parts and move the pointers inward.
- If no match is found, the remaining substring is taken as a single part and the search ends.
- If one character remains after all matches, count it as an additional part.

Algorithm Steps:

1. Read the number of test cases.
2. For each string, determine the maximum length among all inputs for precomputation.
3. Precompute powers of a large base modulo MOD1 and MOD2 up to that maximum length.
4. For each test string:
   - Compute prefix hashes for the entire string under both moduli.
   - Use two pointers: one starting from the beginning, the other from the end of the string.
   - For every possible substring length d, check if the prefix $s[i : i+d]$ equals the suffix $s[j-d+1 : j+1]$ using precomputed hashes.
   - If a match is found, increment the part count by 2 and move inward.

- If no match is found, take the middle as one part and terminate.
- If one character remains, count it as a final part.

5. Output the number of parts for each test case.

Code Structure

```
main():
    Read number of test cases (t)
    Read all t strings and determine the max length

    Set BASE, MOD1, MOD2 constants for rolling hash
    Precompute powers of BASE modulo MOD1 and MOD2 up to max length

    For each string:
        - Compute prefix hashes for both MOD1 and MOD2
        - Initialize two pointers i = 0 and j = n - 1
        - Initialize part counter to 0

        While i < j:
            - Try all d from 1 to (j - i + 1) // 2
            - Compute hash of prefix and suffix of length d
            - If hashes match under both moduli:
                - Increment parts by 2
                - Move i forward and j backward by d
                - Break loop to restart from new positions

        If no match found in above loop:
            - Add 1 to parts (middle chunk)
            - Return immediately

        If i == j (single char remains):
            - Add 1 to parts

    Output part count for each string
```

Time Complexity:

- Precomputation of powers: $O(n)$
- Hash computation per test: $O(n)$
- Substring hash comparisons per test: amortized $O(n)$

Overall Time Complexity: $\mathbf{O}(\sum |\mathbf{s}|)$ across all test cases
Space Complexity: $\mathbf{O}(\mathbf{n})$ for prefix hashes and power arrays

References

- https://codeforces.com/blog/entry/60445
- https://cp-algorithms.com/string/string-hashing.html
- https://stackoverflow.com/questions/4008541/how-to-split-a-string-into-as-few-palindromes-as-possi

1.1 Note on Double Hashing Moduli

Choosing

$$\text{MOD}_1 = 10^9 + 7, \quad \text{MOD}_2 = 10^9 + 9$$

provides the following benefits:

- **Large prime moduli**: Ensures uniform distribution of hash values and avoids cycle patterns.
- **Collision resistance**: Requiring both hashes to match reduces the probability of a false positive to about 10^{-18}.
- **Adversarial safety**: Two independent primes make it practically impossible for crafted test cases to force a hash collision.
- **Efficient arithmetic**: Both primes fit safely within 64-bit integer ranges, allowing fast and overflow-free computations.

2 C. Happy Dog, Happy Life: Codeforces Submission ID: 320834652

Explanation:

We are given a sequence of toys, each with a "funness" value. The goal is to maximize the total happiness of a dog who plays with these toys. The dog is very picky i.e he only accepts toys in strictly increasing order of funness. However, if he accepts a toy on day i, he continues playing with it for all subsequent days. The total happiness is defined as the sum of the funness values of all toys the dog has on each day.

For example, if toy i has funness f_i and is given on day i, it contributes f_i for all days from i to n, where n is the total number of toys. Therefore, its contribution is $f_i \times (n - i)$. The task becomes selecting a strictly increasing sequence of toys such that the total contribution (i.e., sum of such weighted funness values) is maximized.

Plan of Attack:

This problem is a variant of the classical **Weighted Longest Increasing Subsequence (Weighted LIS)**. A naive $O(n^2)$ solution would not work for $n \leq 10^5$, so an efficient $O(n \log n)$ approach using a Fenwick Tree is implemented.

- For each toy, compute its weighted contribution as $f[i] \times (n - i)$.
- To avoid dealing with large funness values directly in the Fenwick Tree, compress the values into a continuous range using coordinate compression.
- Use a Fenwick Tree (Binary Indexed Tree) to store and query the maximum DP value (total happiness so far) for all smaller funnesses.
- Iterate through the toys:
  - For each toy i, query the maximum total happiness so far among all funness values strictly less than $f[i]$.
  - Add the current toy's weight to this maximum to compute the best possible total happiness ending at toy i.
  - Update the Fenwick Tree with this value.
- Keep track of the global maximum across all DP values.

Algorithm Steps:

1. Read the number of toys and their funness values.
2. Compute the weight of each toy as $f[i] \times (n - i)$.
3. Coordinate compress the funness values to the range $[1 \dots m]$.
4. Initialize a Fenwick Tree of size m.
5. For each toy i:
   - Let `idx` be the compressed index of $f[i]$.
   - Query the maximum DP value from the Fenwick Tree up to `idx - 1`.
   - Compute `dp[i]` as that max value plus the toy's weight.
   - Update the Fenwick Tree at `idx` with `dp[i]`.
6. Track the global maximum DP value as the final answer.

Code Structure

```
main():
    - Read number of toys (n)
    - Read list of funness values f[0..n-1]
    - Compute weight[i] = f[i] * (n - i)
    - Coordinate compress all funness values to indices 1..m

    - Initialize Fenwick Tree of size m
    - Initialize answer = 0

    For each toy i from 0 to n-1:
        - Get compressed index idx of f[i]
        - Query BIT[1..idx-1] to find max previous DP value
        - dp[i] = best_prev + weight[i]
        - Update BIT[idx] with dp[i]
        - Update global answer = max(answer, dp[i])

    Print answer
```

Time Complexity:

- Weight computation: $O(n)$
- Coordinate compression: $O(n \log n)$
- DP using Fenwick Tree: $O(n \log n)$

Overall Time Complexity: $\mathbf{O(n \log n)}$
Space Complexity: $\mathbf{O(n)}$ for weights, compressed mapping, and Fenwick Tree

References

- `https://cp-algorithms.com/data_structures/fenwick.html`
- `https://www.geeksforgeeks.org/fenwick-tree-for-competitive-programming/`
- `https://stackoverflow.com/questions/8782943/maximum-weight-increasing-subsequence`

3 D. Colloquium: Codeforces Submission ID: 320835281

Explanation:

We are given a sequence of scheduled talks, each categorized as either `T` (Theory), `S` (Systems), or `A` (AI). These talks must be divided into two sections in such a way that the total student attention time is maximized. The attention time for each talk depends on the diversity of the last three talks in the respective section:

- 20 minutes if all three belong to the same category,
- 40 minutes if two different categories are present,
- 60 minutes if all three are from distinct categories.

The key constraint is that talks must be assigned in the original order, and each talk must be assigned to one (and only one) section.

Plan of Attack:

This problem is solved using dynamic programming (DP), where the state of the DP is defined based on the recent history of talks in both sections.

- Encode each category as a numerical ID for convenience.
- Maintain a DP dictionary, where each state is a 4-tuple representing the two most recent categories in each section.
- For every incoming talk, iterate through all current DP states and attempt to assign the talk to either of the two sections.
- For each assignment:
  - Update the section's history.
  - Calculate the new attention time based on the diversity of the last three talks in that section.
  - Normalize the state to handle symmetric equivalence (i.e., avoid counting permutations of the same history multiple times).
  - Update the new DP state with the best achievable attention time.
- After processing all talks, the maximum attention value across all DP states gives the final answer.

Algorithm Steps:

1. Read the number of talks and the talk sequence.
2. Encode categories as integers.
3. Initialize the DP with an empty state (no history in either section).
4. For each character in the sequence:
   - For every current DP state:
     - Attempt to assign the talk to section 1 and compute the updated state and attention.
     - Attempt the same for section 2.
     - Normalize states (sort the histories) to avoid duplicates.
     - Update the new DP dictionary with the best value for each unique state.
5. After all talks are processed, return the maximum value from the DP table.

Code Structure

```
main():
    Read number of talks (n) and the string of categories

    Map each category to a numerical ID
    Initialize a DP dictionary with the base state (no history)

    For each character in the string:
        For every current DP state:
            - Try assigning the talk to section 1:
                - Update history
                - Calculate attention using diversity of 3 categories
                - Normalize state representation
                - Update new DP with max attention

            - Try assigning to section 2 (same as above)

    After processing all talks, output the maximum attention value
```

Time Complexity:

- Each talk processes a fixed number of possible states (since there are only 3 categories): $O(n)$
- State transitions are limited and efficiently handled via dictionaries

Overall Time Complexity: O(n), where n is the number of talks
Space Complexity: O(1) per step due to bounded state size, **O(n)** overall

References

- https://stackoverflow.com/questions/71539851/dynamic-programming-problem-maximize-the-sum-of-the-v