# 2900. Longest Unequal Adjacent Groups Subsequence I

Medium Greedy Array String Dynamic Programming

Leetcode Link

## Problem Description

The goal is to find the longest subsequence from an array of indices `[0, 1, ..., n - 1]` such that for any two consecutive indices in the subsequence `i_j` and `i_(j + 1)`, the elements in the binary array `groups` at those indices are not the same, i.e., `groups[i_j] != groups[i_(j + 1)]`. Each index in the subsequence corresponds to a word in the `words` array. The task is to return an array of words that represents this longest subsequence.

A subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements. Importantly, the words in the `words` array may have different lengths, which doesn't impact the selection of the subsequence.

## Intuition

The approach to finding the longest subsequence where consecutive elements in `groups` are different is a greedy one. This means we can make local, optimal choices at each step without needing to consider the rest of the array.

For every element at index `i` in `groups`, we have two scenarios - either `i` is the first index (i.e., `i == 0`), or `groups[i]` is different from the previous element `groups[i - 1]`. If either of these conditions is met, we can include the corresponding word `words[i]` in our subsequence.

By iterating over the entire `groups` array and including words that meet the criteria, we ensure that no two consecutive selected words have their corresponding `groups` elements equal, effectively giving us the longest subsequence by the definition provided.

The intuition comes from the fact that to maximize the length of the subsequence, we want to include as many words as possible as long as they meet the aforementioned condition. Since the condition only depends on the current and previous `groups` elements, we only need a simple iteration to build our solution without needing to backtrack or look ahead.

## Solution Approach

The provided Python solution uses a list comprehension to create and return the subsequence of words, which is essentially a single-pass greedy algorithm. The algorithm iterates through the `groups` array and applies a selection criteria to each element to determine if the corresponding element from the `words` array should be included in our final output or not.

**Algorithm:**

1. Initialize a list comprehension that will evaluate each element `x` in `groups` along with its corresponding index `i`. It uses Python's built-in `enumerate` function to obtain each element in `groups` and its index simultaneously.
2. In this comprehension, for every element `x` and index `i`, the following condition is checked: `i == 0 or x != groups[i - 1]`. This condition says that the first element (`i == 0`) should always be included, and then every subsequent element should be included only if it is different than the one preceding it (`x != groups[i - 1]`). This ensures that no two adjacent elements in the subsequence have the same `groups` value.
3. If the condition is true for a given `i`, we select `words[i]` for inclusion in the final output.
4. Once the list comprehension finishes iterating over all elements in `groups`, it will have produced a list of words that constitutes the longest subsequence satisfying the problem's constraints.
5. The final step is to return this list of words.

**Data Structures:**

- We utilize Python's list data structure to store the `words` and `groups`.

**Patterns used:**

- The solution applies a **greedy approach** to the problem: at each step, it makes a local optimum choice (whether to include a word) based only on the current and immediate previous elements from `groups`, which guarantees the finding of the global optimum (the longest subsequence under the given conditions).
- **List comprehension**, a concise way to create lists, is used for its readability and efficiency in selecting the appropriate words.

This simple yet effective approach leverages the characteristics of the problem's constraints to arrive at an optimal solution without needing a complex algorithm.

### Example Walkthrough

Let's walk through a small example to illustrate the solution approach described above. Suppose we have the following `groups` and `words` arrays:

- `groups = [1, 0, 0, 1, 0, 1]`
- `words = ["apple", "banana", "grape", "cherry", "mango", "peach"]`

Our task is to find the longest subsequence such that no two consecutive indices in the subsequence correspond to equal elements in the `groups` array. Let's apply the algorithm step by step:

1. Start iterating through the `groups` array, comparing the element at index `i` with the one at index `i - 1`.
2. The first element in `groups` is `1` (at index `0`). Since `i == 0`, we don't have a previous element to compare with, so we include "apple" from the `words` array in our subsequence.
3. The next element in `groups` is `0` (at index `1`). Since `groups[1] != groups[0]`, we include "banana" from the `words` array in our subsequence.
4. At index `2`, `groups` has another `0`. This time, `groups[2] == groups[1]`, so "grape" does not get included in our subsequence since consecutive elements must be different.
5. Now at index `3`, we have a `1` in `groups`. Since `groups[3] != groups[2]`, "cherry" gets included in our subsequence.
6. Moving to index `4`, the element in `groups` is `0`. Because `groups[4] != groups[3]`, we include "mango" in our subsequence.
7. Lastly, at index `5`, `groups` contains a `1`. As `groups[5] != groups[4]`, we include "peach" in our subsequence.

Following the steps of our algorithm, the final subsequence of `words` is:

- `["apple", "banana", "cherry", "mango", "peach"]`

Thus, by iterating through each element of the `groups` array and checking our defined condition, we successfully construct the longest subsequence of words without having identical consecutive elements from `groups`. This illustrates the effectiveness of the greedy approach to solve the problem, as we made local optimal selections to achieve a global optimal solution.

## Python Solution

```python
# Import the List type from typing module for type hints
from typing import List

class Solution:
    def getWordsInLongestSubsequence(self, n: int, words: List[str], groups: List[int]) -> List[str]:
        # Initialize an empty list to store the words in the longest subsequence
        longest_subsequence_words = []

        # Iterate through each index and corresponding group number in the groups list
        for i, group_number in enumerate(groups):
            # Check if it's the first word or if the current group number is different from the previous one
            if i == 0 or group_number != groups[i - 1]:
                # If yes, append the corresponding word to the longest_subsequence_words list
                longest_subsequence_words.append(words[i])

        # Return the list of words that form the longest subsequence
        return longest_subsequence_words

```

## Java Solution

```java
import java.util.ArrayList;
import java.util.List;

class Solution {

    /**
     * Finds the words in the longest subsequence with alternating groups.
     *
     * @param n      the number of words.
     * @param words  an array of words.
     * @param groups an array of group identifiers corresponding to each word.
     * @return a list of words in the longest subsequence with alternating groups.
     */
    public List<String> getWordsInLongestSubsequence(int n, String[] words, int[] groups) {
        // Initialize an ArrayList to store the resulting words.
        List<String> result = new ArrayList<>();

        // Iterate over the words to find the longest subsequence.
        for (int i = 0; i < n; ++i) {
            // Add the first word and any word that starts a new group (compared to the previous word).
            if (i == 0 || groups[i] != groups[i - 1]) {
                result.add(words[i]);
            }
        }
        // Return the list of words in the longest subsequence.
        return result;
    }
}

```

## C++ Solution

```cpp
#include <vector>
#include <string>

class Solution {
public:
    // Function that generates a vector of strings, which consists of the words
    // in the longest non-repeating subsequence based on the given groups.
    // Parameters:
    // n - the number of elements in the words and groups vectors.
    // words - a vector of strings representing the words.
    // groups - a vector of integers, where each integer corresponds to the group of the word at the same index.
    std::vector<std::string> getWordsInLongestSubsequence(int n, std::vector<std::string>& words, std::vector<int>& groups) {
        // Answer vector to store the resulting words.
        std::vector<std::string> answer;

        // Iterate through each group by index.
        for (int index = 0; index < n; ++index) {
            // If we are at the first word, or the current word belongs to a different group than the previous one,
            // then it is a part of the longest non-repeating subsequence.
            if (index == 0 || groups[index] != groups[index - 1]) {
                // Add the current word to the answer vector.
                answer.emplace_back(words[index]);
            }
        }
        // Return the answer vector containing the words in the longest non-repeating subsequence.
        return answer;
    }
};

```

## Typescript Solution

```typescript
function getWordsInLongestSubsequence(totalWords: number, wordsArray: string[], wordGroups: number[]): string[] {
    // Initialize an array to hold the resulting longest subsequence of words.
    const longestSubsequence: string[] = [];

    // Iterate through the array of words to identify the longest subsequence.
    for (let index = 0; index < totalWords; ++index) {
        // If we are at the first word or the current word's group is different from the previous word's group,
        // it is a part of the longest subsequence, so we add it to the result array.
        if (index === 0 || wordGroups[index] !== wordGroups[index - 1]) {
            longestSubsequence.push(wordsArray[index]);
        }
    }

    // Return the longest subsequence found.
    return longestSubsequence;
}

```

## Time and Space Complexity

The time complexity of the code is `O(n)`, where `n` is the length of the list `groups`. This is because the code iterates over the list `groups` once and performs a constant time check for each element.

The space complexity is `O(k)`, where `k` is the number of unique subsequences identified, which in the worst case could be equal to `n`. This would happen if no two consecutive elements in `groups` are the same, resulting in each word from `words` being added to the output list. Thus, the space used for the output list is proportional to the number of selected words.