1898. Maximum Number of Removable Characters

Medium Array String Binary Search Leetcode Link

Problem Description

You are provided with two strings `s` and `p`, where `p` must be a subsequence of `s`. There is also an array called `removable` with distinct indices from `s`. The goal is to find out the maximum number of characters you can remove from `s` (using the indices provided in `removable` from lowest to highest) so that `p` remains a subsequence of `s`. The steps for removing characters are as follows: mark characters in `s` at indices indicated by the first `k` elements of `removable`, remove them, and then check if `p` remains a subsequence of the modified `s`. A subsequence is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements.

Intuition

The key to solving this problem lies in understanding that once you remove characters from `s`, you must check whether each subsequent removal maintains `p` as a subsequence. Since `removable` contains distinct indices, and we want to maximize `k`, we can approach this problem using binary search.

The intuition behind using binary search here is that if removing `k` characters still leaves `p` as a subsequence, removing fewer than `k` characters will also leave `p` as a subsequence. Conversely, if `p` is not a subsequence after removing `k` characters, removing more than `k` characters will certainly not leave `p` as a subsequence. Therefore, there is a threshold value for `k` which we want to find - the maximum `k` for which `p` remains a subsequence. Binary search allows us to efficiently home in on this threshold by halving the search space with each iteration and identifying the exact point where `p` stops being a subsequence.

In the provided solution, the function `check(k)` takes a number `k` and simulates the process of marking and removing `k` characters from `s` to verify if `p` remains a subsequence. The binary search then adjusts the search interval (`left`, `right`) based on the outcomes of `check(k)`. If `p` is still a subsequence after removing `k` characters, it means we could possibly remove more, so the search space is adjusted to `[mid, right]`. If not, we have removed too many characters, and the search space is adjusted to `[left, mid-1]`. The search continues until the maximum `k` is found.

Solution Approach

The implementation makes use of a binary search pattern, specifically what's often referred to as "Binary Search Template 2". This is due to the way the `mid` point is calculated and how the search space is adjusted. The pattern is conducive when we want to search for the element or condition which requires accessing the current index and its immediate right neighbour in the array.

Here's the breakdown of the solution approach:

1. **Binary Search Algorithm:** We want to find the maximum value of `k` such that removing `k` characters specified by the first `k` indices in `removable` from `s` leaves `p` as a subsequence. We perform a binary search over the length of the array `removable` because the problem has a "yes" or "no" nature (is `p` still a subsequence after the removal?) and because the removal condition is monotonic—that is, if removing `k` characters works, then removing any less than `k` will also work.

2. **`check` Function:** This function simulates the removal of characters from `s` and checks if `p` is still a subsequence. It uses two pointers, `i` for the string `s` and `j` for the string `p`. As it iterates over `s`, it skips over the characters at the indices specified in the first `k` entries of `removable`. If a non-removable character in `s` matches the current character in `p`, it moves the pointer `j` forward. If the end of `p` is reached (`j == n`), then `p` is still a subsequence of `s` after the removals.

3. **Set Data Structure:** A set called `ids` is used to store the indices from `removable` that we're proposing to remove. This allows constant-time checks to determine if a character at a particular index in `s` should be skipped during the subsequence check.

4. **Calculating `mid`:** The variable `mid` is calculated as `(left + right + 1) // 2` to ensure that the search space moves towards the higher numbers when the `check(mid)` is successful, ultimately helping to identify the upper bound of `k`.

5. **Adjusting Search Space:** Based on the results of `check(mid)`, the binary search narrows the search space. If removing `mid` characters works, it means we should look for a possibly higher `k`, so the new search interval is `[mid, right]`. If not, we search the space `[left, mid - 1]`. The search ends when `left` and `right` meet, which will be the point just before `p` stops being a subsequence, thus giving us the maximum `k`.

The combination of these components results in a solution that efficiently pinpoints the largest `k` for which `p` remains a subsequence in `s`, despite the removals.

Example Walkthrough

Let's consider a small example to illustrate the solution approach.

Suppose:

- `s = "abcacb"`
- `p = "ab"`
- `removable = [3, 1, 0]`

Applying Binary Search

1. **Initial Setup:** We set up our binary search with bounds `left = 0` and `right = len(removable) - 1 = 2`. The goal is to find the maximum `k` such that `p` is still a subsequence of the modified `s` after the removal.

2. **First Iteration:**

   - Calculate `mid = (left + right + 1) // 2 = (0 + 2 + 1) // 2 = 1`.
   - Use `check(1)` to simulate the removal of characters at indices `removable[0]` and `removable[1]`, which are `0` and `1`.
   - After removal, `s = "_b_acb"`. `p = "ab"` is still a subsequence of this new string `s`.
   - Since `check(1)` is true, we adjust the search range to `[mid, right]` which means `[1, 2]`.

3. **Second Iteration:**

   - Calculate `mid = (1 + 2 + 1) // 2 = 2`.
   - Use `check(2)` to simulate the removal of characters at indices `removable[0]`, `removable[1]`, and `removable[2]`, which are `0`, `1`, and `3`.
   - After removal, `s = "_b__cb"`. `p = "ab"` is no longer a subsequence of `s`.
   - Since `check(2)` is false, we adjust the search range to `[left, mid - 1]` which means `[1, 1]`.

Since `left` and `right` are the same, we conclude the binary search. We determined that the maximum `k` for which `p = "ab"` remains a subsequence of `s` after removal is `1`.

Therefore, the maximum number of characters that can be removed from `s` while ensuring `p` remains a subsequence is `1` (removing the characters at indices `[0, 1]` from `s`).

Python Solution

```python
from typing import List

class Solution:
    def maximum_removals(self, string: str, pattern: str, removable: List[int]) -> int:
        def can_form_pattern_after_removals(k: int) -> bool:
            # Initialize pointers for pattern and string
            pattern_index = 0
            string_index = 0
            # Convert the list of removable indices up to k into a set for faster look-ups
            removal_set = set(removable[:k])
            # Iterate through string and pattern
            while pattern_index < pattern_length and string_index < string_length:
                # Check if current index is not removed, and characters match
                if string_index not in removal_set and string[string_index] == pattern[pattern_index]:
                    # Move to the next character in the pattern
                    pattern_index += 1
                # Move to the next character in the string
                string_index += 1
            # We can form the pattern if we have gone through all characters of the pattern
            return pattern_index == pattern_length

        # Get string and pattern lengths
        string_length, pattern_length = len(string), len(pattern)
        # Set the search space for the binary search
        left, right = 0, len(removable)
        # Perform binary search to find the maximum number of removable characters
        while left < right:
            mid = (left + right + 1) // 2  # Pick the middle value
            # Check if the pattern can be formed after removing mid characters
            if can_form_pattern_after_removals(mid):
                left = mid  # If it is possible, try removing more characters
            else:
                right = mid - 1  # Otherwise, try with fewer removable characters
        return left  # The maximum number of removable characters is left

# Explanation:
# The above code checks the maximum number of indices that can be removed from the string `s`
# such that it is still possible to find the pattern `p` as a subsequence.
# It uses binary search to efficiently find this maximum number.

```

Java Solution

```java
class Solution {
    // Method to find the maximum number of characters that can be removed from
    // string s so that string p is still a subsequence of s
    public int maximumRemovals(String s, String p, int[] removable) {
        int left = 0, right = removable.length; // Define binary search boundaries

        // Perform binary search to find the maximum index 'mid'
        while (left < right) {
            int mid = (left + right + 1) / 2; // Choose a middle point (avoid overflow)
            // Check if string p is still a subsequence of s after removing mid characters
            if (checkSubsequence(s, p, removable, mid)) {
                left = mid; // Move left pointer to mid if p is still a subsequence
            } else {
                right = mid - 1; // Move right pointer if p is not a subsequence
            }
        }
        return left; // Returns maximum removables that still keep p as a subsequence of s
    }

    // Helper method to check if p is a subsequence of s after removing certain characters
    private boolean checkSubsequence(String s, String p, int[] removable, int mid) {
        int sLen = s.length(); // Length of string s
        int pLen = p.length(); // Length of string p
        int i = 0; // Index for iterating over s
        int j = 0; // Index for iterating over p

        // Create a hash set to store the indices of characters to be removed
        Set<Integer> removalSet = new HashSet<>();
        for (int k = 0; k < mid; k++) {
            removalSet.add(removable[k]); // Add removable characters' indices up to mid
        }

        // Iterate over string s and p to check if p is a subsequence
        while (i < sLen && j < pLen) {
            // If current index is not in the removal set and characters match, move j
            if (!removalSet.contains(i) && s.charAt(i) == p.charAt(j)) {
                j++;
            }
            i++; // Always move i to the next character
        }

        // If we've found the whole string p, return true; otherwise, return false
        return j == pLen;
    }
}

```

C++ Solution

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

class Solution {
public:
    int maximumRemovals(std::string s, std::string p, std::vector<int>& removable) {
        int left = 0, right = removable.size();

        // Binary search to find the maximum number of characters that can be removed
        while (left < right) {
            // Choose the midpoint of the current range
            int mid = (left + right + 1) / 2;

            // Check if the subsequence 'p' is still in 's' after removing 'mid' characters
            if (checkSubsequence(s, p, removable, mid)) {
                left = mid; // If it is, increase the lower bound of the search range
            } else {
                right = mid - 1; // If not, decrease the upper bound of the search range
            }
        }
        return left; // The maximum number of characters we can remove
    }

    // Helper function to check if 'p' is a subsequence of 's' after removing 'mid' characters
    bool checkSubsequence(std::string& s, std::string& p, std::vector<int>& removable, int mid) {
        int sLength = s.size();
        int pLength = p.size();
        int i = 0, j = 0; // Pointers for string 's' and the subsequence 'p'

        // Using a set to keep track of the indices that have been removed
        std::unordered_set<int> removedIndices;
        for (int k = 0; k < mid; ++k) {
            removedIndices.insert(removable[k]);
        }

        // Iterate over both strings and check if 'p' is a subsequence of 's'
        while (i < sLength && j < pLength) {
            // If the current character is not removed and matches with `p[j]`,
            // move the pointer `j` of subsequence `p`
            if (removedIndices.count(i) == 0 && s[i] == p[j]) {
                ++j;
            }
            ++i; // Always move the pointer `i` of string `s`
        }

        // If we have iterated through the entire subsequence `p`, it's a subsequence of `s`
        return j == pLength;
    }
};

```

Typescript Solution

```typescript
// Function to determine the maximum number of characters that can be removed
// from string 's' such that string 'p' is still a subsequence of 's'.
function maximumRemovals(s: string, p: string, removable: number[]): number {
    let leftIndex = 0;
    let rightIndex = removable.length;

    while (leftIndex < rightIndex) {
        // Midpoint calculation to use as the possible number of characters to be removed
        let midPoint = Math.floor((leftIndex + rightIndex + 1) / 2);

        // Check if 'p' is still a subsequence after removing 'midPoint' characters
        if (isSubsequence(s, p, new Set(removable.slice(0, midPoint)))) {
            leftIndex = midPoint;
        } else {
            rightIndex = midPoint - 1;
        }
    }
    // The maximum number of characters that can be removed
    return leftIndex;
}

// Function to check if 'sub' is a subsequence of 'str' when indices in 'removedIndices' are removed.
function isSubsequence(str: string, sub: string, removedIndices: Set<number>): boolean {
    let strLength = str.length;
    let subLength = sub.length;
    let indexStr = 0;
    let indexSub = 0;

    // Iterate over each character of 'str' and 'sub' to check for subsequence
    while (indexStr < strLength && indexSub < subLength) {
        // If the current index is not removed, and characters match, increment 'sub' index
        if (!removedIndices.has(indexStr) && str.charAt(indexStr) === sub.charAt(indexSub)) {
            indexSub++;
        }
        indexStr++;
    }
    // Check if the end of 'sub' was reached, indicating it is a subsequence
    return indexSub === subLength;
}

```

Time and Space Complexity

Time Complexity:

The time complexity of the function mainly comes from two parts: the binary search and repeatedly checking whether `p` is a subsequence of `s` after certain characters have been removed.

The binary search is done over the length of `removable`, which has a complexity of `O(log k)` where `k` is the length of `removable`. For each iteration of binary search, the `check` function is called, which iterates over the characters of `s` and `p`.

The `check` function can take `O(m + k)` time in the worst case, where `m` is the length of `s` and `k` is the length of the prefix of `removable` considered in the check function. Although `k` changes with each binary search iteration, we need to consider the worst-case scenario where the function checks the entire prefix, hence `k`.

Combining these two parts, the overall time complexity is `O((m + k) * log k)`.

Space Complexity:

The space complexity is primarily due to the set `ids` used in the `check` function, which stores the indices that are removable up to `k`. This set can grow up to size `k`.

Thus, the space complexity is `O(k)`.