942. DI String Match

Easy Greedy Array Two Pointers String Leetcode Link

Problem Description

The problem presents a scenario where we need to construct a permutation of integers from `0` to `n` based on a given string `s` of length `n`. Each character in the string represents a relationship between consecutive numbers in the permutation: 'I' indicates that the precedent integer should be less than the subsequent integer, and 'D' dictates the reverse, meaning the precedent should be greater than the subsequent. Our goal is to construct any valid permutation that satisfies the conditions represented by the string `s`.

Intuition

To solve this problem, we utilize a two-pointer approach to keep track of the smallest and largest numbers not yet placed in the permutation. Initially, we set `low` to `0` and `high` to `n`, representing the lowest and highest possible values in the permutation. We iterate through the string `s`, and based on whether the current character is an 'I' or a 'D', we either place the `low` or `high` value in the permutation and then update `low` or `high` accordingly.

- When we encounter an 'I', it indicates that `perm[i]` should be less than `perm[i + 1]`, so we can safely place the current `low` value at this position and increment `low` to the next smallest available number.
- When we encounter a 'D', it implies that `perm[i]` should be greater than `perm[i + 1]`, so we place the current `high` value at this position and decrement `high` to the next largest available number.

After processing all characters of the string `s`, we're left with only one number—either the current `low` or `high` value—both of which are equal at this point. We append this last number to the permutation to complete it.

This approach guarantees that the constructed permutation will fulfill the requirements dictated by the string, and since we exhaust all numbers from `0` to `n`, it also ensures that the generated permutation is complete and valid.

Solution Approach

The given Python solution employs a greedy algorithm that builds the permutation incrementally. Here is a breakdown of how the solution is implemented:

- Initialize two pointers, `low` starting at `0` and `high` starting at `n`. These pointers represent the smallest and largest numbers not yet used in the permutation, respectively.
- Create an empty list `ans` to collect the elements of the permutation.
- Loop through each character in the input string `s` with index `i` ranging from `0` to `n - 1`.
  - If `s[i]` is `'I'`, it means the current element of the permutation (indexed by `i`) should be less than the next element:
    - Append `low` to the permutation (`ans.append(low)`).
    - Increment `low` to the next smallest unused number (`low += 1`).
  - If `s[i]` is `'D'`, it means the current element should be greater than the next:
    - Append `high` to the permutation (`ans.append(high)`).
    - Decrement `high` to the next largest unused number (`high -= 1`).
- After the loop, there will be one number left, which will be equal to both `low` and `high` since they should have converged. Append this number to `ans`.
- Return the list `ans`, which now contains the valid permutation.

The algorithm's correctness is guaranteed because each step conforms to the description stated in the problem: appending `low` on encountering an `'I'` results in the next element being greater, and appending `high` on encountering a `'D'` results in the next element being smaller. By incrementing/decrementing `low` and `high`, the algorithm also makes sure that each number from `0` to `n` is used exactly once as required for a permutation.

This methodology leverages a simple yet effective pattern that uses available information at each step to make an optimal choice without needing to consider future elements. The use of array or list data structures for storing the permutation is a natural fit for the problem since permutations are, by definition, ordered collections of elements.

Example Walkthrough

Let's consider an example where `n = 3` and `s = "ID"`. The goal is to create a permutation of integers `0` to `3` that satisfies the pattern described by `s`.

Following the solution approach:

- We begin with two pointers, `low` set to `0` and `high` set to `3`.
- We start with an empty permutation list `ans`.

Now, let's loop through the string `s`:

- For `i = 0`, the first character of `s` is 'I', which means that the current element should be less than the next element.
  - We append `low` (which is `0`) to `ans`, now `ans = [0]`.
  - We increment `low` to `1`.
- For `i = 1`, the next character is 'D', so the current element should be greater than the next.
  - We append `high` (which is `3`) to `ans`, now `ans = [0, 3]`.
  - We decrement `high` to `2`.

At this point, we've exhausted `s`, but we have one more position in the permutation to fill (the `n`-th position, considering 0-indexing). Since both `low` and `high` point to `2`, we append the number `2` to `ans`. Now, `ans = [0, 3, 2]`.

Finally, we have one remaining number, which is `1`. We append it to the end of the permutation `ans`, resulting in `ans = [0, 3, 2, 1]`. This is a valid permutation that satisfies the condition 'ID': 0 is less than 3 (`I`), and 3 is greater than 2 (`D`).

The final permutation `ans = [0, 3, 2, 1]` is a valid result, guaranteeing the satisfaction of 'I' and 'D' constraints placed by string `s`.

Python Solution

```python
from typing import List

class Solution:
    def diStringMatch(self, S: str) -> List[int]:
        # Determine the length of the given string S
        n = len(S)

        # Initialize two pointers, one starting at 0 (low) and one starting at n (high)
        low, high = 0, n

        # Create an empty list to store the answer
        answer = []

        # Loop through each character in the string
        for char in S:
            # If the current character is 'I', append the current low value to answer
            # and increment low
            if char == 'I':
                answer.append(low)
                low += 1
            # If the current character is 'D', append the current high value to answer
            # and decrement high
            else: # char == 'D'
                answer.append(high)
                high -= 1

        # After the loop, there will be one remaining element, which is low (or high)
        # Append it to the answer list
        answer.append(low) # at this point, low == high

        # Return the constructed answer list
        return answer

```

Java Solution

```java
class Solution {
    public int[] diStringMatch(String S) {
        // Determine the length of the input string
        int length = S.length();

        // Create variables for the lowest and highest possible values
        int low = 0, high = length;

        // Initialize the answer array of length input+1 (to include all numbers from 0 to length)
        int[] answer = new int[length + 1];

        // Loop through each character in the input string
        for (int i = 0; i < length; i++) {
            // If the current character is 'I', assign `low` to the current position and increment `low`
            if (S.charAt(i) == 'I') {
                answer[i] = low++;
            }
            // If the current character is 'D', assign `high` to the current position and decrement `high`
            else {
                answer[i] = high--;
            }
        }

        // After looping through the string, the `low` and `high` should be equal, assign it to the last position
        answer[length] = low; // It could also be `high` as both will have the same value

        // Return the computed permutation of integers
        return answer;
    }
}

```

C++ Solution

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

class Solution {
public:
    // Function to generate permutations according to the DI string pattern
    std::vector<int> diStringMatch(std::string str) {
        int length = str.size(); // Get the size of the input string
        int low = 0; // Initialize the lowest possible value
        int high = length; // Initialize the highest possible value
        std::vector<int> result(length + 1); // Initialize the result vector with size length + 1

        // Iterate through each character in the input string
        for (int i = 0; i < length; ++i) {
            // If the current character is 'I', assign the lowest available number and increment 'low'
            if (str[i] == 'I') {
                result[i] = low++;
            } else { // If the current character is not 'I' (thus 'D'), assign the highest available number and decrement 'high'
                result[i] = high--;
            }
        }

        // Since each 'D' and 'I' in the string consumed a number (low or high), the last number left is 'low'
        result[length] = low; // Assign the last element, which will be equal to 'low' (or 'high' since they now should be the same)

        return result; // Return the resulting permutation vector
    }
};

```

Typescript Solution

```typescript
function diStringMatch(s: string): number[] {
    // The length of the input string
    const length = s.length;
    // Resultant array which will hold the permutation of length+1 elements
    const result = new Array(length + 1);
    // Initialize pointers for the lowest and highest possible values
    let low = 0;
    let high = length;

    // Iterate over the string characters
    for (let i = 0; i < length; i++) {
        // If the current character is 'I', assign the lowest value and increase it
        if (s[i] === 'I') {
            result[i] = low++;
        // If the current character is not 'I' (hence 'D'), assign the highest value and decrease it
        } else {
            result[i] = high--;
        }
    }

    // Assign the last element in the result array,
    // which is either the increased low or the decreased high (after the loop, they are equal)
    result[length] = low;
    // Return the resulting permutation array
    return result;
}

```

Time and Space Complexity

The code defines a function `diStringMatch` that takes a string `s` and generates a permutation of integers from `0` to `len(s)` such that adjacent elements correspond to 'I' (increase) and 'D' (decrease) in the string `s`.

Time Complexity

The function consists of a single loop that iterates over the string `s` exactly once, with a constant number of operations performed during each iteration. The number of iterations is equal to `n` where `n` is the length of the string `s`. After the loop, one additional element is appended to the `ans` list. However, this does not change the overall time complexity. Therefore, the time complexity of the function is `O(n)`.

Space Complexity

The function uses extra space in the form of the list `ans` which will contain `n + 1` elements by the end of the function's execution, as every character in the string `s` corresponds to an element being added to the list, with one final element being added after the loop completes. Thus, the space complexity of the function is also `O(n)`.