484. Find Permutation

Medium Stack Greedy Array String

Problem Description

string s maps to a permutation perm of length n + 1, where n is the length of s. The characters in the string s determine whether the consecutive numbers in the permutation perm are in increasing (I) or decreasing (D) order. Our goal is to reconstruct the permutation perm from the given sequence s such that the permutation is the lexicographically

In this LeetCode problem, we are given a string s that represents a sequence of I (increase) and D (decrease) operations. This

smallest possible. Lexicographically smallest means that if we treat each permutation as a number, we want the smallest possible number that can be formed while still satisfying the pattern of I and D from the string s. For example, if the string s is IID, then a possible permutation that fits the pattern is 12345 where perm[0] < perm[1], perm[1]

< perm[2] and perm[3] > perm[4]. However, 12345 may not be the lexicographically smallest permutation, and our task is to find the smallest one.

right. This means that whenever we encounter an I in the string s, we should choose the smallest available number at that point

Intuition

to keep the permutation small. Conversely, when we encounter a D, we want to defer choosing the numbers until we reach the end of the consecutive sequence of Ds, and then fill in the numbers in reverse. By doing so, we ensure that the number before the D sequence is greater than the numbers after it, while still keeping those numbers as small as possible. The process to arrive at this solution involves the following steps: Fill in the array ans with the numbers from 1 to n+1 in ascending order. Assuming initially that all the following characters are

Traverse the string s and when we see a D, identify the consecutive sequence of Ds. We note where this sequence ends.

The intuition behind the solution comes from the fact that we want to make the permutation as small as possible from left to

Reverse the sub-array ans[i : j + 1] where i is the start of the D sequence, and j is the index of the last D in the

'I', this would be the lexicographically smallest permutation.

Continue this process until we have gone through the entire string s.

sequence. This places the larger numbers before the smaller numbers, which is required by the D sequence.

- By following these steps, we construct the lexicographically smallest permutation that satisfies the given sequence s.
- Solution Approach

number from 1 to n + 1 exactly once, this fills in the default ascending order for the "I" scenario.

The implementation of the solution follows the intuition closely, using mostly basic data structures and control flow patterns

Here's a step-by-step breakdown of the algorithm implemented by the Solution class: Initialize a list called ans that contains the numbers from 1 to n + 1 (inclusive). Because the sequence must contain each

pattern given by s.

available in Python.

While i is less than the length of the string s, look for consecutive "D"s. This is achieved by initializing another pointer j to i, which moves forward as long as it finds "D".

For each sequence of "D"s, reverse the corresponding subsequence in lans. The slice lans[i : j + 1] represents the

Update the pointer i to continue from the end of the handled "D" sequence. The max(i + 1, j) ensures that if j has not

numbers that need to be in descending order to satisfy this sequence of "D"s. By reversing using the [::-1] slice, we arrange them correctly while keeping them as small as possible.

Create a pointer i that will traverse the string s from the beginning.

- moved (indicating that there were no "D"s), i continues to the next character. If j has moved, i skips over the "D" sequence. After the completion of the loop, and now represents the lexicographically smallest permutation in accordance with the
- "D"s. Once the list ans has been fully traversed and modified, it is returned as the lexicographically smallest permutation that follows the increase-decrease pattern dictated by string s. By using this straightforward implementation, there is no need for complex

This algorithm leverages Python's list indexing and slicing capabilities to reverse subarrays efficiently. The in-place reversal of

subarrays helps to maintain the overall lexicographic order by ensuring that the smallest values are placed after sequences of

complexity stays constrained to O(n), where n is the number of elements in perm. **Example Walkthrough**

First, we need to initialize a list ans that contains numbers from 1 to n + 1. Since the length of s is 3, n + 1 equals 4.

data structures or additional space beyond the initial list to store the permutation. The in-place operations ensure that the space

Now we start traversing the string s. At s[0], we have D, which signifies that ans[0] should be greater than ans[1]. We find the next sequence of "I" or the end of the string to determine the range to reverse. In this case, the next character at s[1] is I, signaling the end of our D sequence. So, we have a D sequence from i = 0 to j = 0. We reverse ans[i:j+1], but in this case, since i and j are the same, the list remains [1, 2, 3, 4].

We move to the next character in the string s and increment i to $\max(i + 1, j)$, which makes i = 1.

At s[1], we have I, so we leave ans as it is because the permutation should remain in ascending order for I.

We move i to 2 because there was no D sequence.

• Initial ans: [1, 2, 3, 4]

Solution Implementation

from typing import List

index = 0

Python

class Solution:

sequence is from i = 2 to j = 2.

At s[2], we have D again, so we look for the next I or the end of the string. The end of the string comes next, so our D

string s. Here's a step by step representation of ans after each iteration:

• After handling I at s[1]: [1, 2, 3, 4] (no change since ans is already ascending)

• After handling D at s[0]: [1, 2, 3, 4] (no change since it's a single D)

• After handling D at s[2]: [1, 2, 4, 3] (the last two numbers are reversed)

Let's illustrate the solution approach with a simple example where the string s is DID.

We then create a pointer i starting at the beginning of the string s. Initially, i = 0.

Therefore, we initialize ans to [1, 2, 3, 4].

We reverse ans [i : j + 1], which means reversing the slice [3, 4]. After the reversal, ans becomes [1, 2, 4, 3]. Iteration is complete as we reach the end of s. 10.

Now, the list ans represents [1, 2, 4, 3], which is the lexicographically smallest permutation following the DID pattern of the

The final result is the permutation [1, 2, 4, 3], which satisfies the original pattern and is lexicographically the smallest

- possible.
- def findPermutation(self, s: str) -> List[int]: # Length of the input pattern $pattern_length = len(s)$

Start iterating over the pattern

sequence_end += 1

while index < pattern length:</pre>

sequence end = index

print(solution.findPermutation("DID"))

Create an initial list of integers from 1 to n+1

Find the end of the current 'D' sequence

index = max(index + 1, sequence_end)

public int[] findPermutation(String pattern) {

int[] result = new int[length + 1];

for (int i = 0; i <= length; ++i) {</pre>

int length = pattern.length();

permutation = list(range(1, pattern_length + 2))

For the pattern 'DI', the initial list will be [1, 2, 3]

while sequence end < pattern_length and s[sequence_end] == 'D':</pre>

Reverse the sub-list corresponding to the 'D' seguence found

This is done because numbers must be in descending order for 'D'

permutation[index : sequence_end + 1] = permutation[index : sequence_end + 1][::-1]

Move to next starting point, one past this 'D' sequence or increment by one

// This function produces the permutation of numbers based on the given pattern string.

// Initialize the result array with natural numbers starting from 1 to n+1

int currentIndex = 0; // Start from the beginning of the string.

// Find the sequence of 'D's in the input string.

currentIndex = max(currentIndex + 1, nextIndex);

return permutation; // Return the resulting permutation.

// Helper function to generate an increasing sequence array from 1 to n

return Array.from({ length: n }, (_, index) => index + 1);

// Helper function to reverse a subarray from start to end indices

function reverse(array: number[], start: number, end: number): void {

const n = sequence.length; // Get the size of the input sequence

Return the modified list as the resulting permutation

[array[start], array[end]] = [array[end], array[start]]; // swap elements

// Initialize the permutation array as an increasing sequence from 1 to n+1

while (nextIndex < n && sequence[nextIndex] == 'D') {</pre>

++nextIndex; // Move to the next index if it's a 'D'.

// Reverse the subvector from the start of 'D's to just past the last 'D'.

reverse(permutation.begin() + currentIndex, permutation.begin() + nextIndex + 1);

// Move to the index after the sequence of 'D's or increment by one if no 'D's were found.

// Iterate through the entire sequence.

int nextIndex = currentIndex;

while (currentIndex < n) {</pre>

function iota(n: number): number[] {

while (start < end) {</pre>

start++;

```
# Return the modified list as the resulting permutation
return permutation
```

Java

Example usage:

class Solution {

solution = Solution()

```
result[i] = i + 1;
        int index = 0;
        // Traverse through the pattern string
        while (index < length) {</pre>
            int startIndex = index;
            // Find the contiquous sequence of 'D's
            while (startIndex < length && pattern.charAt(startIndex) == 'D') {</pre>
                startIndex++;
            // Reverse the sequence to fulfill the 'D' requirement in permutation
            reverse(result, index, startIndex);
            // Advance the index to the position after the reversed section or move it at least one step forward.
            index = Math.max(index + 1, startIndex);
        return result;
    // This function reverses the elements in the array between indices i and j.
    private void reverse(int[] array, int start, int end) {
        // Note end is decremented to swap the elements inclusively
        for (int left = start, right = end - 1; left < right; ++left, --right) {</pre>
            // Swap elements at indices left and right
            int temp = array[left];
            array[left] = array[right];
            array[right] = temp;
C++
#include <vector>
#include <algorithm>
#include <numeric>
class Solution {
public:
    // Method to find the permutation according to the input sequence.
    vector<int> findPermutation(string sequence) {
        int n = sequence.size(); // Size of the input string
        // Create a vector to store the answer, initializing
        // it with an increasing sequence from 1 to n+1.
        vector<int> permutation(n + 1);
        iota(permutation.begin(), permutation.end(), 1);
```

end--; // Function to find the permutation as per the input sequence function findPermutation(sequence: string): number[] {

};

TypeScript

```
let permutation = iota(n + 1);
  let currentIndex = 0; // Start from the beginning of the sequence
  // Iterate through the entire sequence
 while (currentIndex < n) {</pre>
    let nextIndex = currentIndex;
    // Find the sequence of 'D's in the input string
    while (nextIndex < n && sequence[nextIndex] === 'D') {</pre>
      nextIndex++; // Advance to the next index if it's a 'D'
    // Reverse the subarray from the start of 'D's to just past the last 'D'
    reverse(permutation, currentIndex, nextIndex);
    // Move to the index after the sequence of 'D's or increment by one if no 'D's were found
    currentIndex = Math.max(currentIndex + 1, nextIndex);
  return permutation; // Return the resulting permutation
from typing import List
class Solution:
    def findPermutation(self, s: str) -> List[int]:
       # Length of the input pattern
        pattern_length = len(s)
       # Create an initial list of integers from 1 to n+1
       # For the pattern 'DI', the initial list will be [1, 2, 3]
        permutation = list(range(1, pattern_length + 2))
       # Start iterating over the pattern
        index = 0
        while index < pattern length:</pre>
           # Find the end of the current 'D' sequence
            sequence end = index
           while sequence end < pattern_length and s[sequence_end] == 'D':</pre>
                sequence_end += 1
            # Reverse the sub-list corresponding to the 'D' sequence found
            # This is done because numbers must be in descending order for 'D'
            permutation[index : sequence_end + 1] = permutation[index : sequence_end + 1][::-1]
           # Move to next starting point, one past this 'D' sequence or increment by one
            index = max(index + 1, sequence_end)
```

Time Complexity

return permutation

print(solution.findPermutation("DID"))

Time and Space Complexity

The time complexity of the code snippet is O(n), where n is the length of the input string s. This is because the code involves

0(n).

Example usage:

solution = Solution()

iterating over each character in the input string up to twice. The while loop iterates over each character, and within this loop, there is another while loop that continues as long as the character is 'D'. However, the inner loop advances the index j every time it finds a 'D', thereby skipping that section for the outer loop. Therefore, even though there is a nested loop, each character will be looked at a constant number of times, resulting in linear time complexity overall. Additionally, the reversal of the sublist ans[i : j + 1] takes linear time with respect to the length of the sublist, but since it's done without overlapping with other sublists, the total amount of work for all reversals summed together still does not exceed

Space Complexity The space complexity is 0(n) because we are storing the result in a list ans which contains n + 1 elements. No additional

significant space is used, and the space used for the output does not count towards extra space complexity when analyzing space complexity. The in-place reversal operation does not incur additional space cost.