2899. Last Visited Integers

Easy Array String Simulation

Leetcode Link

Problem Description

The problem provides us with an array of strings where each element is either a string that represents a positive integer or the string "prev". We're required to iterate through this array and, whenever we encounter the string "prev", we must find the last visited integer according to specific rules.

For a sequence of one or more "prev" strings, the number of consecutive "prev" strings (including the current "prev") is counted as k. We have to look at the integers seen so far in reverse order. The last visited integer is the (k-1)th integer in this reversed order. If k exceeds the total number of integers we've seen, then the last visited integer is to be considered -1.

Our goal is to return an array of integers, which includes the last visited integer for each "prev" string found in the input array.

For example, given an input array of `["1", "2", "prev", "prev", "3", "prev"]`, our output should be `[2, 1, -1]`.

Intuition

The intuition behind the solution is to simulate the process described, keeping track of two essential pieces of information: the integers we have seen (in order) and the current sequence length of "prev" strings (k).

To achieve this, we can keep an array called `nums` to store all the integers we have encountered so far in the order they were visited. We also keep a count `k`, which is reset to 0 every time we encounter an integer and is incremented when we encounter a "prev". Whenever we bump into a "prev", we look k elements back from the end of our stored integers—if there are enough elements—and add that to the result array. If there are not enough elements (k is greater than the number of integers encountered), we add -1 to the result.

Essentially, we are creating a stack of numbers encountered, and for every "prev", we are performing a lookup that acts like indexing from the back of the stack by k positions, simulating the process of tracking the last visited integers.

Solution Approach

The solution follows a simple simulation approach that revolves around keeping track of the visited integers and the number of consecutive occurrences of "prev". This approach can be outlined as follows:

1. We initialize an array `nums` to store integers that have been seen. This array acts as a stack where we can easily add new integers and look up past integers.
2. A variable `k` is used to keep count of consecutive "prev" strings. It is reset to 0 whenever a non-"prev" string (i.e., an integer) is encountered.
3. We iterate through the `words` array, processing one string at a time.
4. If the current word is not "prev" (it is a positive integer), we reset `k` to 0 and append the integer form of the word to `nums`.
5. If the current word is "prev", we increment `k` by 1 to account for the new "prev" string in the sequence.
6. We then attempt to retrieve the last visited integer by looking `k` places from the end of `nums` using the index `len(nums) - k`. If `k` is within the range of `nums`, we append the desired integer to `ans`; otherwise, we append `-1`, indicating that there are not enough visited integers to satisfy the "prev" condition.
7. After processing all words in the array, we return the `ans` array, which contains the last visited integer for each "prev" encountered.

The solution uses straightforward array manipulation and conditions to achieve the objective, employing basic data structures (lists in Python), and index manipulation. The primary pattern this solution leverages is iteration with condition checks, ensuring the last visited integer is correctly identified and handled according to the problem's rules.

Here's a breakdown of the code corresponding to the steps above:

```python
class Solution:
    def lastVisitedIntegers(self, words: List[str]) -> List[int]:
        nums = []  # Step 1: initialize the stack of seen integers
        ans = []   # Initialize the array for storing last visited integers
        k = 0      # Initialize the count of consecutive "prev"
        for w in words:  # Step 3: iterate through each word
            if w == "prev":  # Step 4 and 5: handle "prev"
                k += 1  # Increment count for consecutive "prev"
                i = len(nums) - k  # Calculate index for the last visited integer
                ans.append(-1 if i < 0 else nums[i])  # Append the last visited integer or -1
            else:  # Step 6: reset k and add new integer to nums
                k = 0
                nums.append(int(w))  # Convert string to integer and add to nums
        return ans  # Step 7: return the result
```

Example Walkthrough

Let's go through an example to illustrate the solution approach using a small input array: `["10", "prev", "20", "prev", "prev"]`.

1. Initialize `nums` as an empty array to represent the stack of seen integers, `ans` as an empty array for the last visited integers, and `k` as 0 for the count of consecutive "prev" strings.
2. Iterate through each element in `words`:
   - For the first element `"10"`, since it is not "prev", reset `k` to 0 (it's already 0). Convert `"10"` to an integer and append it to `nums`. Now `nums = [10]`.
   - The second element is `"prev"`. Increment `k` to 1 (k was 0). Calculate the index: `len(nums) - k` which is `0`. There is an available integer, so append `nums[0]` (which is `10`) to `ans`. Now `ans = [10]`.
   - The third element `"20"` is not "prev", so reset `k` to 0 and append `20` to `nums`. Now `nums = [10, 20]`.
   - The fourth element is again `"prev"`. Increment `k` to 1. Calculate index `len(nums) - k` which is `1`. Append `nums[1]` (which is `20`) to `ans`. Now `ans = [10, 20]`.
   - The last element is `"prev"` again. Increment `k` to 2 (since we do not reset `k` as the element is "prev"). Calculate index `len(nums) - k` which is `0`. Append `nums[0]` (which is `10`) to `ans`. Now `ans = [10, 20, 10]`.
3. Finally, we end up with `ans = [10, 20, 10]`, which is the output, with each entry representing the last visited integer for each "prev".

By processing each element one by one and keeping track of the seen integers (`nums`) and the consecutive count of "prev" (`k`), we can efficiently simulate the process and determine the last visited integer or `-1` when required.

Python Solution

```python
class Solution:
    def lastVisitedIntegers(self, words: List[str]) -> List[int]:
        # Initialize a list to keep track of the integers seen so far.
        seen_numbers = []
        # Initialize a list to keep track of the outputs for each "prev" command.
        output = []
        # Counter to keep track of how many "prev" commands have been seen consecutively.
        prev_count = 0
        # Iterate through each word in the words list.
        for word in words:
            if word == "prev":
                # Increment the 'prev' counter if the current word is "prev".
                prev_count += 1
                # Compute the index of the integer to access based on 'prev_count'.
                index = len(seen_numbers) - prev_count
                # Append the number to the output if it exists, otherwise append -1.
                output.append(-1 if index < 0 else seen_numbers[index])
            else:
                # Reset 'prev_count' to 0 since the current word is a number.
                prev_count = 0
                # Convert the word to an integer and append it to the seen_numbers.
                seen_numbers.append(int(word))
        # Return the output list which contains the integers or -1 for each "prev".
        return output

```

Java Solution

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

class Solution {

    // Method to find the last visited integers based on given words
    public List<Integer> lastVisitedIntegers(List<String> words) {
        // A list to store the actual numbers seen so far.
        List<Integer> numbers = new ArrayList<>();

        // A list to store the result of previously visited numbers.
        List<Integer> result = new ArrayList<>();

        // Variable to keep track of how many 'prev' operations have been encountered.
        int prevCount = 0;

        // Iterate over each word in the input list.
        for (String word : words) {
            if ("prev".equals(word)) {
                // If the word is 'prev', increment the counter and get the last visited number.
                prevCount++;
                int index = numbers.size() - prevCount; // Calculate the index for previously visited number.
                if (index < 0) {
                    // If the index is out of bounds, add -1 to the result.
                    result.add(-1);
                } else {
                    // Otherwise, add the number at the calculated index to the result.
                    result.add(numbers.get(index));
                }
            } else {
                // If the word is a number, reset the counter and add the number to our number list.
                prevCount = 0;
                numbers.add(Integer.valueOf(word));
            }
        }

        // Return the result list containing the last visited numbers.
        return result;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>
using namespace std;

class Solution {
public:
    // Function to process a list of strings and return a vector representing the last visited integers
    vector<int> lastVisitedIntegers(vector<string>& words) {
        vector<int> nums; // Vector to store parsed integers from the 'words' vector
        vector<int> ans;  // Vector to store answers for each "prev" command
        int prevCounter = 0; // Counter to keep track of the "prev" commands

        // Iterate over the words
        for (auto& word : words) {
            // If the current word is "prev", find the previously encountered integer
            if (word == "prev") {
                ++prevCounter; // Increment counter for each "prev"
                int index = nums.size() - prevCounter; // Calculate the index for the previously visited integer

                // If the index is valid, push the found integer to the 'ans' vector; otherwise push -1
                ans.push_back(index < 0 ? -1 : nums[index]);
            } else {
                // Reset the prevCounter and add the numeric value of the word to the 'nums' vector
                prevCounter = 0;
                nums.push_back(stoi(word));
            }
        }

        return ans; // Return the result vector
    }
};

```

Typescript Solution

```typescript
function lastVisitedIntegers(words: string[]): number[] {
    // Initialize a list to keep track of numeric inputs
    const numberList: number[] = [];
    // Initialize a list for the answer which will hold the last visited integers
    const answerList: number[] = [];
    // Initialize a counter to keep track of the "prev" commands
    let prevCounter = 0;

    // Iterate through each word in the input array
    for (const word of words) {
        // Check if the current word is the 'prev' command
        if (word === 'prev') {
            // Increment the counter as we've encountered a 'prev'
            ++prevCounter;
            // Calculate the index we want to access
            const indexToAccess = numberList.length - prevCounter;
            // Check if the index is valid, if not, push -1 to answerList
            answerList.push(indexToAccess < 0 ? -1 : numberList[indexToAccess]);
        } else {
            // Reset the counter since a number is encountered
            prevCounter = 0;
            // Convert the string to a number and push to the numberList
            numberList.push(Number(word));
        }
    }

    // Return the answer list containing all last visited integers on encountering 'prev'
    return answerList;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n)`, where `n` is the length of the `words` array. We iterate over each word in the array once. In the iteration, the operations we perform, such as checking the value of `w` and updating the `nums` list or the index `k`, all have constant time complexity (`O(1)`). Even when we access `nums[i]` the time complexity is `O(1)` because list indexing is a constant time operation in Python. Hence, the main contributing factor to the time complexity is the single loop through the `words` array, resulting in `O(n)` time complexity.

Space Complexity

The space complexity of the code is also `O(n)`. We use additional data structure `nums` to store the integers as they appear when they are not "prev". Thus, in the worst-case scenario where there is no "prev", `nums` will have the same number of integers as there are elements in the `words` list. The `ans` list at most will contain `n` "-1" integers when all elements in `words` are "prev". The sum of the size of `nums` and `ans` lists dictates the space complexity. Therefore, the space complexity is `O(n)` due to the storage requirements that scale linearly with the input size.