946. Validate Stack Sequences

Medium Stack Array Simulation Leetcode Link

Problem Description

In this problem, we are given two integer arrays named `pushed` and `popped` respectively. Each array contains distinct values. The aim is to determine whether the sequence of integers in the `popped` array could be the result of a certain sequence of push and pop operations on an initially empty stack by only using the integers given in the `pushed` array, or not. To clarify, we want to know if we can push the numbers from the `pushed` array onto the stack in the order they are given, and then pop them off to match the order they appear in the `popped` array. We are to return `true` if this is possible, or `false` otherwise.

Intuition

The intuition behind the solution is to simulate the push and pop operations of a stack using the given `pushed` and `popped` arrays.

1. We traverse the `pushed` array and push each element onto the stack.
2. After each push, we check if the stack's top element is equal to the current element at index `j` of the `popped` array.
3. If it is, we pop that element from the stack and increase the `j` index to move to the next element in the `popped` array.
4. We continue this process until either the stack is empty or the stack's top element is not equal to the `popped[j]`.
5. The stack simulates the push and pop operations, and the index `j` keeps track of the number of elements correctly popped from the stack.
6. If, after all push operations, the `j` index equals the length of the `pushed` array, it means all elements were popped in the `popped` order, so we return `true`.
7. If `j` is not equal to the length of `pushed` after the simulation, it means the `popped` sequence is not possible with the given push and pop operations, thus we return `false`.

The key insight is to recognize that the stack allows us to reorder elements as they are pushed and popped, but the `popped` array creates a constraint on the order in which elements must be popped. The solution algorithm effectively tries to match these constraints by simulating the stack operations.

Solution Approach

The solution uses a simple stack data structure and a loop to simulate the stack operations. Below is a step-by-step explanation of the solution:

1. Initialize an empty list `stk` that will act as our stack, and a variable `j` to keep track of the current index in the `popped` array.

```python
j, stk = 0, []
```

2. Begin a loop to iterate over each value `v` in the `pushed` array.

```python
for v in pushed:
```

3. Inside the loop, push the current element `v` onto the stack (`stk`).

```python
stk.append(v)
```

4. After the push operation, enter a `while` loop that will run as long as the stack is not empty and the top element of the stack is equal to the element at the current index `j` of the `popped` array. This checks whether we can pop the top element to match the next element to be popped according to the `popped` sequence.

```python
while stk and stk[-1] == popped[j]:
```

5. If the top element on the stack is the same as `popped[j]`, pop it from the stack and increment `j` to check against the next element in the `popped` array. This simulates the pop operation and progresses the index as matching elements are found and popped.

```python
stk.pop()
j += 1
```

6. After the loop has iterated through all elements in `pushed`, check if the index `j` is now the same as the length of the `pushed` array. If it is, it means all `pushed` elements have been successfully matched with the `popped` elements in the right order, so the function returns `true`.

```python
return j == len(pushed)
```

7. If `j` is not equal to `len(pushed)`, it means not all elements could be matched, so the sequence of operations is not possible, and the function returns `false`.

In this solution, the stack data structure is essential since it allows elements to be accessed and removed in a last-in, first-out manner, which is exactly what we need to simulate the push and pop operations. The `while` loop within the `for` loop ensures that as soon as an element is pushed onto the stack, it is checked to see if it can be immediately popped off to follow the `popped` sequence. This continues the checking and popping of elements without pausing the push operations, effectively interleaving the necessary operations to achieve the end goal.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Consider the following `pushed` and `popped` arrays:

```python
pushed = [1, 2, 3, 4, 5]
popped = [4, 5, 3, 2, 1]
```

Follow the steps outlined in the solution approach:

1. Initialize an empty stack `stk` and `j` index to `0`.
2. Traverse the `pushed` array:
   - `v = 1`: Push 1 into `stk`. `stk = [1]`. Since `stk[-1]` (1) is not equal to `popped[0]` (4), we continue without popping.
   - `v = 2`: Push 2 into `stk`. `stk = [1, 2]`. Since `stk[-1]` (2) is not equal to `popped[0]` (4), we continue without popping.
   - `v = 3`: Push 3 into `stk`. `stk = [1, 2, 3]`. Since `stk[-1]` (3) is not equal to `popped[0]` (4), we continue without popping.
   - `v = 4`: Push 4 into `stk`. `stk = [1, 2, 3, 4]`. Since `stk[-1]` (4) is equal to `popped[0]` (4), we pop from `stk` and increment `j`. After popping, `stk = [1, 2, 3]` and `j = 1`.
3. Continue iterating and check the top against `popped[j]`. The new values of `j` and `popped` we compare against are as follows:
   - Now `stk[-1]` is 3 and `popped[j]` is 5. They are not the same, so we move to push the next element from `pushed`.
   - `v = 5`: Push 5 into `stk`. `stk = [1, 2, 3, 5]`. Now, `stk[-1]` is 5, which matches `popped[j]` (5), so we pop 5 from `stk`, and now `stk = [1, 2, 3]` and increment `j` to 2.
   - Now `stk[-1]` is 3, which matches `popped[j]` (3), so we pop 3 from `stk`, and now `stk = [1, 2]` and increment `j` to 3.
   - Continue this comparison and popping process:
     - Pop 2 from `stk` because it matches `popped[j]` (2), and now `stk = [1]` and increment `j` to 4.
     - Pop 1 from `stk` because it matches `popped[j]` (1), and now `stk` is empty and increment `j` to 5.
4. Since we've finished iterating through `pushed`, we check if `j` is now equal to the length of `pushed` (which is 5):
   - Indeed, `j` is 5, which is the length of `pushed`, suggesting that all elements were popped in the `popped` order.

Hence, for the given `pushed` and `popped` arrays, the simulation shows that it is possible to match the push and pop sequence, and the function should return `true`.

Python Solution

```python
from typing import List

class Solution:
    def validateStackSequences(self, pushed: List[int], popped: List[int]) -> bool:
        # Initialize index to track the position in the 'popped' sequence
        pop_index = 0

        # Initialize an empty list to simulate stack operations
        stack = []

        # Iterate through each value in the 'pushed' sequence
        for value in pushed:
            # Push the current value onto the stack
            stack.append(value)

            # Check if the top of the stack matches the current value in 'popped'
            # If it does, pop from the stack and advance the index in 'popped'
            while stack and stack[-1] == popped[pop_index]:
                stack.pop()
                pop_index += 1

        # If the pop_index is equal to the length of 'pushed', all elements were popped
        # in the correct order, hence we return True. Otherwise, return False.
        return pop_index == len(pushed)

# Example usage:
# solution = Solution()
# result = solution.validateStackSequences([1, 2, 3, 4, 5], [4, 5, 3, 2, 1])  # True
# result = solution.validateStackSequences([1, 2, 3, 4, 5], [4, 3, 5, 1, 2])  # False

```

Java Solution

```java
class Solution {

    // Method to validate stack sequences using provided pushed and popped array sequences
    public boolean validateStackSequences(int[] pushed, int[] popped) {
        // Use a Deque as a stack for simulating the push and pop operations
        Deque<Integer> stack = new ArrayDeque<>();

        // Index for keeping track of the position in the popped sequence
        int popIndex = 0;

        // Iterate over the pushed sequence to simulate the stack operations
        for (int num : pushed) {
            // Push the current number onto the stack
            stack.push(num);

            // Keep popping from the stack if the top of the stack matches the current
            // number in the popped sequence
            while (!stack.isEmpty() && stack.peek() == popped[popIndex]) {
                stack.pop(); // Pop from stack
                popIndex++;  // Move to the next index in the popped sequence
            }
        }

        // If all elements were successfully popped in the correct sequence, the popIndex
        // should be equal to the length of the pushed sequence
        return popIndex == pushed.length;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <stack>

class Solution {
public:
    // This function checks whether a given stack push and pop sequence is valid
    bool validateStackSequences(vector<int>& pushed, vector<int>& popped) {

        // Initialize an empty stack
        stack<int> stack;

        // The index for the popped sequence
        int popIndex = 0;

        // Iterate over each value in the pushed sequence
        for (int value : pushed) {
            // Push the current value onto the stack
            stack.push(value);

            // While the stack is not empty and the top of the stack is equal to
            // the next value in the popped sequence
            while (!stack.empty() && stack.top() == popped[popIndex]) {
                // Pop the top value off the stack
                stack.pop();
                // Increment the pop sequence index
                popIndex++;
            }
        }

        // If the popIndex equals the size of the pushed sequence,
        // then all elements were popped in the correct order
        // Hence, the sequences are valid and the method returns true.
        // If elements remain in the stack or the popIndex does not reach the end,
        // then the sequences are not valid and the method returns false.
        return popIndex == pushed.size();
    }
};

```

Typescript Solution

```typescript
function validateStackSequences(pushed: number[], popped: number[]): boolean {
    // Initialize an empty array to simulate stack operations
    const stack: number[] = [];
    // Index to keep track of the position in the 'popped' sequence
    let popIndex = 0;

    // Iterate through each value in the 'pushed' sequence
    for (const value of pushed) {
        // Push the current value onto the stack
        stack.push(value);
        // Continue popping from the stack if the top element equals
        // the next element in the 'popped' sequence
        while (stack.length && stack[stack.length - 1] === popped[popIndex]) {
            stack.pop(); // Remove the top element from the stack
            popIndex++; // Move to the next index in the 'popped' sequence
        }
    }

    // If all elements were successfully popped in the 'popped' sequence order,
    // then the popIndex should match the length of the 'pushed' array
    return popIndex === pushed.length;
}

```

Time and Space Complexity

Time Complexity: The time complexity of the code is `O(n)`, where `n` is the length of the `pushed` and `popped` lists. The for loop runs for each element in `pushed`, and while each element is pushed onto the stack once, the while loop may not necessarily run for every push operation as it depends on the match with the `popped` sequence. However, each element will be popped at most once. Therefore, each element from `pushed` is involved in constant-time push and pop operations on the stack, making the time complexity linear.

Space Complexity: The space complexity of the code is `O(n)`. In the worst case, all the elements from the `pushed` list could be stacked up in the `stk` array (which happens when the `popped` sequence corresponds to reversing the `pushed` sequence), which would take up a space proportional to the number of elements in `pushed`.