

946. Validate Stack Sequences

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.

- We traverse the `pushed` array and push each element onto the stack.
- After each push, we check if the stack's top element is equal to the current element at index `j` of the `popped` array.
- 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.
- We continue this process until either the stack is empty or the stack's top element is not equal to the `popped[j]`.
- The stack simulates the push and pop operations, and the index `j` keeps track of the number of elements correctly popped from the stack.
- 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`.
- 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:

- 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.

```
1 j, stk = 0, []
```
- Begin a loop to iterate over each value `v` in the `pushed` array.

```
1 for v in pushed:
```
- Inside the loop, push the current element `v` onto the stack (`stk`).

```
1 stk.append(v)
```
- 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.

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

```
1 stk.pop()
2 j += 1
```
- 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`.

```
1 return j == len(pushes)
```
- If `j` is not equal to `len(pushes)`, 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:

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

Follow the steps outlined in the solution approach:

- Initialize an empty stack `stk` and `j` index to 0.
- 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`.
- 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.
- 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

```
1 from typing import List
2
3 class Solution:
4     def validateStackSequences(self, pushed: List[int], popped: List[int]) -> bool:
5         # Initialize index to track the position in the 'popped' sequence
6         pop_index = 0
7
8         # Initialize an empty list to simulate stack operations
9         stack = []
10
11        # Iterate through each value in the 'pushed' sequence
12        for value in pushed:
13            # Push the current value onto the stack
14            stack.append(value)
15
16            # Check if the top of the stack matches the current value in 'popped'
17            # If it does, pop from the stack and advance the index in 'popped'
18            while stack and stack[-1] == popped[pop_index]:
19                stack.pop()
20                pop_index += 1
21
22        # If the pop_index is equal to the length of 'pushed', all elements were popped
23        # If not the correct order, hence we return True. Otherwise, return False.
24        return pop_index == len(pushes)
25
26 # Example usage:
27 solution = Solution()
28 result = solution.validateStackSequences([1, 2, 3, 4, 5], [4, 5, 3, 2, 1]) # True
29 result = solution.validateStackSequences([1, 2, 3, 4, 5], [4, 3, 5, 1, 2]) # False
30
```

Java Solution

```
1 class Solution {
2
3     // Method to validate stack sequences using provided pushed and popped array sequences
4     public boolean validateStackSequences(int[] pushed, int[] popped) {
5         // Use a Deque as a stack for simulating the push and pop operations
6         Deque<Integer> stack = new ArrayDeque<>();
7
8         // Index for keeping track of the position in the popped sequence
9         int popIndex = 0;
10
11        // Iterate over the pushed sequence to simulate the stack operations
12        for (int num : pushed) {
13            // Push the current number onto the stack
14            stack.push(num);
15
16            // Keep popping from the stack if the top of the stack matches the current
17            // number in the popped sequence
18            while (!stack.isEmpty() && stack.peek() == popped[popIndex]) {
19                stack.pop(); // Pop from stack
20                popIndex++; // Move to the next index in the popped sequence
21            }
22        }
23
24        // If all elements were successfully popped in the correct sequence, the popIndex
25        // should be equal to the length of the pushed sequence
26        return popIndex == pushed.length;
27    }
28 }
29
```

C++ Solution

```
1 #include <vector>
2 #include <stack>
3
4 class Solution {
5 public:
6     // This function checks whether a given stack push and pop sequence is valid
7     bool validateStackSequences(vector<int>& pushed, vector<int>& popped) {
8
9         // Initialize an empty stack
10        stack<int> stack;
11
12        // The index for the popped sequence
13        int popIndex = 0;
14
15        // Iterate over each value in the pushed sequence
16        for (int value : pushed) {
17            // Push the current value onto the stack
18            stack.push(value);
19
20            // While the stack is not empty and the top of the stack is equal to
21            // the next value in the popped sequence
22            while (!stack.empty() && stack.top() == popped[popIndex]) {
23                // Pop the top value off the stack
24                stack.pop();
25                // Increment the pop sequence index
26                popIndex++;
27            }
28        }
29
30        // If the popIndex equals the size of the pushed sequence,
31        // then all elements were popped in the correct order.
32        // Hence, the sequences are valid and the method returns true.
33        // If elements remain in the stack or the popIndex does not reach the end,
34        // then the sequences are not valid and the method returns false.
35        return popIndex == pushed.size();
36    }
37 };
38
```

Typescript Solution

```
1 function validateStackSequences(pushed: number[], popped: number[]): boolean {
2     // Initialize an empty array to simulate stack operations
3     const stack: number[] = [];
4     // Index to keep track of the position in the 'popped' sequence
5     let popIndex = 0;
6
7     // Iterate through each value in the 'pushed' sequence
8     for (const value of pushed) {
9         // Push the current value onto the stack
10        stack.push(value);
11
12        // Continue popping from the stack if the top element equals
13        // the next element in the 'popped' sequence
14        while (stack.length && stack[stack.length - 1] === popped[popIndex]) {
15            stack.pop(); // Remove the top element from the stack
16            popIndex++; // Move to the next index in the 'popped' sequence
17        }
18    }
19
20    // If all elements were successfully popped in the 'popped' sequence order,
21    // then the popIndex should match the length of the 'pushed' array
22    return popIndex === pushed.length;
23 }
```

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`.