1441. Build an Array With Stack Operations

Medium Stack Array Simulation

Leetcode Link

Problem Description

The task involves simulating a stack operation to construct a specific sequence of numbers, given as the `target` array, using a stream of numbers from 1 to `n`. We can only use two operations: "Push", to add the next number in the stream on top of the stack, and "Pop", to remove the top element of the stack.

The goal is to ensure that the numbers in the stack, from bottom to top, match the sequence described in the `target` array. We need to do this by adhering to specific rules:

- We can only take the next integer from the stream if the stream isn't empty and push it onto the stack.
- We can pop the top integer from the stack if the stack isn't empty.
- We must stop taking new integers from the stream and stop performing operations on the stack once it matches the `target`.

The expected output is a list of the stack operations ("Push" and "Pop") needed to attain the `target` sequence.

Intuition

To solve this problem, we need to simulate the process of either pushing or popping elements to match the `target` sequence. We iterate over the `target` array and compare its elements with the next expected integer from the stream, which we represent with a variable (let's say, `cur`). For every number in the `target` array, we do the following:

- If `cur` is less than the current `target` value, it implies that there are numbers in the stream that we need to skip to match the `target`. For each of these numbers, we "Push" (to simulate taking the number from the stream) and then "Pop" (to remove it).
- We increment `cur` until it matches the current `target` value, then perform one "Push" operation without a corresponding "Pop" since this number should stay in the stack.

We continue this process for each element in the `target` array. The sequence of "Push" and "Pop" operations that we accumulate forms the solution to rebuild the `target` array from the stream. It is worth mentioning that multiple valid sequences of operations can exist, and we can return any of them.

This approach works because it effectively simulates the steps needed to recreate the `target` array using the given stack operations, mirroring the actual manual process one might use if doing this with a physical stack and stream of numbers.

Solution Approach

The solution takes a simple iterative approach to build the required sequence of stack operations. Here is a step-by-step explanation of the implementation details:

1. Initialize two variables: `cur`, which represents the next expected number from the stream starting from 1, and `ans`, which holds the list of stack operations performed.
2. Loop through each value `v` in the `target` array:
   - Inside this loop, increment `cur` by 1 - it means we are considering the push of the next number in the stream.
   - While `cur` is less than the current `target` value `v`, we realize that we are not interested in having `cur` in our final stack. Therefore, we perform a "Push" followed immediately by a "Pop". Add these two operations to the `ans` list, and then increment `cur` to consider the next number in the stream.
   - By the time `cur` matches `v`, the current value of the `target` array, we only need to "Push" as we want to keep this number in our stack. Hence, we add a "Push" operation to `ans`.
3. After iterating through each element in the `target` array, we have a list of operations that leads us to a stack that, if evaluated from bottom to top, will match `target`.
4. Return the `ans` list, which contains the sequence of "Push" and "Pop" operations.

This algorithm effectively uses a simulation pattern, where we simulate each step needed to construct the target sequence from the available input numbers (1 to `n`). The `while` loop inside the iteration is essential as it allows us to skip over the numbers that are not included in the `target` sequence by simulating a push/pop pair for each.

Here is how the pattern works with the code:

```
class Solution:
    def buildArray(self, target: List[int], n: int) -> List[str]:
        cur, ans = 0, []
        for v in target:
            cur += 1
            while cur < v:
                ans.extend(['Push', 'Pop'])
                cur += 1
            ans.append('Push')
        return ans
```

In this code:

- The `for` loop iterates over the `target`.
- The `while` loop handles skipping numbers not in `target`.
- The `extend(['Push', 'Pop'])` method is used to simulate the push/pop operations for numbers we wish to skip.
- When the condition of the `while` loop is not met (meaning `cur` equals `v`), it appends a `Push` to `ans`.

By the end of the iteration, `ans` represents the sequence of stack operations that would build `target` from the stream of integers within the range `[1, n]`.

Example Walkthrough

Consider the target array `[2, 4]` and `n = 4`. We want to simulate stack operations to reconstruct this sequence where the numbers in the stream range from 1 to `n`.

Step 1: We initialize `cur` to 0 and `ans` to an empty list. These will keep track of the next expected number from the stream and the list of operations, respectively.

Step 2: We iterate through each value `v` in the target array. The first value is `2`.

- We increment `cur` to 1 and start a while loop that will run since `cur < v`.
- While `cur < 2`, we do a "Push" (as if we took `1` from the stream and added it to the stack) followed by a "Pop" (we remove `1` since it's not in the target). We add these operations to `ans`, resulting in `['Push', 'Pop']`.
- We increment `cur` again, now `cur` is 2 which matches `v`.
- We perform a "Push" as `cur` is equal to `v` to keep `2` on the stack, updating `ans` to `['Push', 'Pop', 'Push']`.

Step 3: We move on to the next value in the target array, which is `4`.

- We increment `cur` to 3 since it is still less than `v` and perform another "Push" followed by "Pop", updating `ans` to `['Push', 'Pop', 'Push', 'Push', 'Pop']`.
- Incrementing `cur` once more, now `cur` is 4 which matches `v`.
- We perform a "Push" to keep 4 on the stack, updating `ans` to `['Push', 'Pop', 'Push', 'Push', 'Pop', 'Push']`.

Step 4: After processing each element in the target array, we have a complete list of operations that forms the output. Thus, the array `['Push', 'Pop', 'Push', 'Push', 'Pop', 'Push']` is the sequence of operations needed to simulate the stack to achieve the target sequence `[2, 4]`.

Following the above steps ensures that we only push numbers onto the stack that are present in the `target` and pop those that are not, using the simulation approach described in the solution.

This example clearly demonstrates the iterative process of the algorithm, incrementing the `cur` variable, and conditionally adding 'Push' and 'Pop' operations to the `ans` list to match the `target` array.

Python Solution

```
class Solution:
    def buildArray(self, target: List[int], n: int) -> List[str]:
        current_value = 1  # Start with the first value in the stack sequence
        operations = []  # Initialize list to keep track of the operations

        # Iterate over each value that needs to be in the target stack
        for target_value in target:
            # Keep pushing and popping until the current value matches the target value
            while current_value < target_value:
                operations.append('Push')  # Add the value to the stack
                operations.append('Pop')  # Immediately remove it since it's not in target
                current_value += 1  # Move to the next value in the sequence

            # Once the current value matches the target value, add it to the stack
            operations.append('Push')
            # Move to the next value after successfully pushing the current target value
            current_value += 1

        # Return the list of operations that builds the target stack
        return operations
```

Java Solution

```
class Solution {
    public List<String> buildArray(int[] target, int n) {
        // Initialize the current number we are at, starting from 0 before the sequence begins.
        int current = 0;

        // Initialize the answer list to store the output sequence.
        List<String> operations = new ArrayList<>();

        // Iterate over the elements of the target array.
        for (int targetValue : target) {
            // Read the next number and compare it to the target value.
            // If the current number is less than the target value, we "Push" and then "Pop".
            while (++current < targetValue) {
                operations.add("Push");  // Append "Push" to signify we pushed the current number.
                operations.add("Pop");   // Append "Pop" to signify we removed the number we just added.
            }
            // After reaching the target value, perform a "Push" to add the target number to stack.
            operations.add("Push");
        }

        // Return the list of operations needed to get the target sequence.
        return operations;
    }
}
```

C++ Solution

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

// The Solution class contains a method buildArray to generate a sequence of "Push" and "Pop"
// operations to build a specific target array from another array that consists of the first 'n' integers.
class Solution {
public:
    // This method takes in a target vector and an integer n and returns a vector of strings representing the required operations.
    vector<string> buildArray(vector<int>& target, int n) {
        // Initialize the current number to track the numbers we've reached.
        int currentNumber = 0;

        // Initialize an answer vector to hold the sequence of operations.
        vector<string> operations;

        // Iterate over each number in the target array.
        for (int targetValue : target) {

            // While the next number to push is less than our target value, emulate a push and a pop (i.e., skip the number).
            while (++currentNumber < targetValue) {
                operations.emplace_back("Push");  // Emulate adding currentNumber to the stack
                operations.emplace_back("Pop");   // Emulate removing it since it's not our target
            }

            // Once we've reached the targetValue, add it to the stack with one "Push".
            operations.emplace_back("Push"); // Add targetValue to the stack.
        }

        // Return the list of operations required to build the target array.
        return operations;
    }
};
```

Typescript Solution

```
// Function to simulate stack operations to build a target array
function buildArray(target: number[], n: number): string[] {
    // Initialize the result array to hold the sequence of operations
    const operations = [];

    // Track the current number to compare with target array
    let currentNumber = 0;

    // Iterate over the target array to determine operations
    for (const targetNumber of target) {
        // Increment current number and append 'Push' and 'Pop' until it matches the target number
        while (++currentNumber < targetNumber) {
            operations.push('Push'); // Simulate pushing the next number onto the stack
            operations.push('Pop');  // Immediately remove it as it's not in the target
        }
        // When current number matches the target number, just perform the 'Push' operation
        operations.push('Push');
    }

    // Return the array of operations
    return operations;
}
```

Time and Space Complexity

The time complexity of the given code can be analyzed based on the number of operations performed relative to the length of `target` list and `n`. Assume the length of `target` is `m`.

For every element in `target`, the code performs a series of "Push" and "Pop" operations until it reaches the current target value. In the worst case, this could happen for each target value (i.e., if the `target` list has all consecutive numbers starting from 1 up to `n`). Since at each step of iterating through the target list you can have at most two operations (one "Push", and if the number is not in the target one "Pop") per number until you reach the target value, the number of these operations is linear with respect to the last value in `target`, denoted as `target[-1]`.

Therefore, the worst-case time complexity is $O(target[-1])$, which in the worst case is $O(n)$ if the last value in `target` is equal to `n`.

The space complexity of the code is determined by the size of the `ans` list, which holds a sequence of strings representing the operations. In the worst-case scenario as described above, the `ans` list will contain a number of elements equal to twice the value of `target[-1]` minus the length of `target` (since each number not in `target` adds two operations, and each number in `target` adds one operation). Thus, the space complexity is also $O(target[-1])$, which in the worst case is $O(n)$.