385. Mini Parser

Medium · Stack · Depth-First Search · String · Leetcode Link

Problem Description

The problem requires writing a parser that can take a string `s` which represents the serialized form of a nested list, and to deserialize it into a `NestedInteger`. A `NestedInteger` can be either a single integer or a nested list where each element can in turn be an integer or another nested list. The input string follows a specific format where integers are represented as usual, lists start with a left square bracket `[`, end with a right square bracket `]`, and are separated by commas `,`. The challenge is to methodically process this string and construct the corresponding `NestedInteger` with the correct nesting of lists and integers.

Intuition

To arrive at a solution for deserializing the nested list, the key realization is that we need to handle two cases differently: when we encounter an integer and when we encounter a nested list. For integers, the process is straightforward as we just need to read the digits (taking into account the sign, if present) and convert them to an integer value. When dealing with nested lists, however, we need to parse the list recursively or maintain a stack to keep track of the current nested list being constructed.

For the solution involving recursion, we observe that a list contains elements separated by commas and each element can either be an integer or a nested list. We need a way to detect the level of nesting we are at, so when we encounter a comma or the end of the current list, we can decide whether to form an integer `NestedInteger` or initiate a nested recursive call for a deeper level list.

Otherwise, for a non-recursive approach using stacks, we can simulate the call stack used in recursion manually. Each time we encounter a new list represented by the `[` character, we create a new `NestedInteger` and push it onto the stack. Numbers are constructed character by character, and when we hit a delimiter like `,` or `]`, we either finalize the number or a sub-list and add it to the `NestedInteger` at the top of the stack. If the delimiter is a `]` and we have more than one `NestedInteger` in the stack, we pop the top one and add it to the new top as its child, effectively building the nested structures from the inside out.

Both approaches take into account the depth when parsing the string and handle the processing of digits (with their potential negative signs) and the initiation of new lists. The recursion approach is more straightforward but can be more challenging to understand at first glance. The stack approach is an iterative translation of the same process, representing the nesting with a stack data structure.

Solution Approach

Algorithm

The problem can be approached using two methods: recursion or iteration with a stack. Both methods will systematically parse the input string and construct the `NestedInteger` object accordingly.

Solution 1: Recursion

With recursion, the function will call itself to handle the complexity of nested structures. Starting at the outermost list and working inwards each time we encounter a new nested list:

- If we come across a number, we create a NestedInteger with this value.
- When a comma or end of the string is found at the base level (depth 0), we know we've finished parsing a list item, so we recursively parse this item.
- If we encounter a left bracket `'['`, we increase our depth to parse a new nested list.
- A right bracket `']'` indicates that we've finished the current nested list, so we decrease our depth.

The recursion takes into consideration the depth we're currently at, managing integers at the current depth by converting them to `NestedInteger` objects and initiating recursive calls for nested lists.

The time complexity is `O(n)` where `n` is the length of the string, as it processes each character once. The space complexity is also `O(n)` accounting for the recursive call stack.

Solution 2: Stack

The stack approach mimics the call stack that would be created by recursion. Here's a step-by-step explanation of our implementation strategy using a stack:

- The stack is initialized with an empty `NestedInteger`.
- Each time we encounter a digit, we build the number by multiplying the current number by `10` and adding the new digit.
- A negative sign sets a boolean flag that will be used to negate the number when it is complete.
- Encountering a left bracket `'['` means a new nested list is started, and an empty `NestedInteger` is pushed onto the stack.
- A right bracket `']'` or comma `','` signifies the end of a number or list. If it's the end of a number, we complete the number and add it to the `NestedInteger` at the top of the stack.
- If it's a right bracket, and the stack has more than one `NestedInteger`, we pop the stack and add the popped `NestedInteger` to the next item as a nested list.

The flow of the stack effectively captures the opening and closing of nested lists, accumulating elements as it parses through the string. Upon reaching the end of the input, the stack will contain a single `NestedInteger` representing the fully constructed nested list.

The stack-based approach also has a time complexity of `O(n)` as it processes each character in the string once, while the space complexity is `O(n)` due to the stack used to simulate the nested levels.

Both methods utilize the `NestedInteger` API functions such as `isInteger()`, `add()`, `setInteger()`, `getInteger()`, and `getList()` to manipulate and check the types of `NestedInteger`.

In both solutions, handling negative numbers and converting string representations of integers to actual integer values are key operations. Additionally, both methods ensure that we construct the `NestedInteger` objects piecemeal, according to the syntactic structure of the input string.

Example Walkthrough

Consider the input serialized string `s = "[123,[456,[789]]]"`. Let's apply the stack approach to see how the `NestedInteger` is constructed step-by-step.

1. Initialize an empty stack and push a new `NestedInteger` onto it. The stack now contains one empty `NestedInteger`, which will eventually be our result.
2. Start parsing the string from the first character:
   - `[` encountered: This indicates the start of a nested list. Push a new `NestedInteger` onto the stack. The stack now has two elements: an empty `NestedInteger` and a new one just pushed.
3. Next, we encounter `123`:
   - Characters `1`, `2`, `3` are digits, so we construct the number `123`. Since the digits came with no preceding negative sign, the number remains positive.
4. Upon encountering the comma `,`, we've reached the end of an integer:
   - Add `123` as a `NestedInteger` to the top `NestedInteger` in the stack.
5. `[` is found after the comma:
   - This signals a new nested list. Push an empty `NestedInteger` onto the stack.
6. Then we see `456`:
   - As with `123`, we parse `456` as an integer and add it as a `NestedInteger` to the top `NestedInteger` of the stack at this point.
7. After `456`, another `[` is encountered:
   - Again, the opening of a new nested list. Push a new `NestedInteger` onto the stack.
8. Now we parse `789`:
   - The digits `7`, `8`, `9` form the integer `789`, which is added as a `NestedInteger` to the top `NestedInteger` on the stack.
9. We encounter the closing bracket `]`:
   - This indicates the end of the most recent nested list. Pop the top `NestedInteger` from the stack (which contains `789`) and add it to the new top `NestedInteger` (which currently contains `456`). The stack has one less nested list now.
10. Another `]` comes after:
    - Similar to the previous step, we pop the top `NestedInteger` (which now contains `456` and the nested `789`) and add it to the new top `NestedInteger`. This results once more in one less nested list on the stack.
11. The final `]` is encountered:
    - We are now at the end of the input string. If there was another `NestedInteger` in the stack, we would pop and add as before; otherwise, this is the terminating step.

By the end of the parsing process, the stack has a single `NestedInteger` containing the entire nested structure equivalent to the input string. In this example, the final `NestedInteger` would represent the nested list structure with `123` at the first level, and the nested list `[456,[789]]` as its child.

The stack approach has allowed us to parse and construct the nested structure without the need for recursion, effectively handling the hierarchies and complexities of the input serialized string.

Python Solution

```python
class Solution:
    def deserialize(self, s: str) -> NestedInteger:
        # If the string does not start with '[', it means it is an integer.
        # We can simply create a NestedInteger with its value.
        if s[0] != '[':
            return NestedInteger(int(s))

        # Initialize a stack to store the nested structure, along with variables
        # to accumulate numbers and a flag for negative numbers.
        stack = []
        num = 0
        is_negative = False

        # Iterate over each character in the input string.
        for i, char in enumerate(s):
            if char == '-':
                # If the current character is '-', the number is negative.
                is_negative = True
            elif char.isdigit():
                # Accumulate the numerical value, considering the place value as well.
                num = num * 10 + int(char)
            elif char == '[':
                # Start of a nested list.
                # Create an empty NestedInteger and push onto the stack.
                stack.append(NestedInteger())
            elif char in ',]':
                # If the previous character was a digit, finalize the current number,
                # add it to the NestedInteger on the top of the stack.
                if s[i - 1].isdigit():
                    if is_negative:
                        num = -num
                    stack[-1].add(NestedInteger(num))
                # Reset the number and is_negative flag for the next number.
                num = 0
                is_negative = False

                if char == ']' and len(stack) > 1:
                    # End of the current nested list.
                    # Pop the topmost NestedInteger from the stack, which represents
                    # the just-ended nested list, and add it to the next NestedInteger
                    # on top of the stack.
                    nested_integer = stack.pop()
                    stack[-1].add(nested_integer)

        # After processing all characters, the top of the stack contains the NestedInteger
        # representing the entire structure. So return it.
        return stack.pop()

```

Java Solution

```java
import java.util.Deque;
import java.util.ArrayDeque;
import java.util.List;

class Solution {

    // Deserializes a string representation of a nested list into a NestedInteger object.
    public NestedInteger deserialize(String s) {
        // If the string starts with an integer, parse it and return a NestedInteger with that value.
        if (s.charAt(0) != '[') {
            return new NestedInteger(Integer.parseInt(s));
        }

        // Initialize a stack to hold the NestedInteger objects.
        Deque<NestedInteger> stack = new ArrayDeque<>();
        int number = 0; // Used to store the current number being processed.
        boolean isNegative = false; // Flag to check if the current number is negative.

        // Iterate through each character in the string.
        for (int i = 0; i < s.length(); ++i) {
            char character = s.charAt(i);
            if (character == '-') {
                // If the current character is a minus sign, set the isNegative flag.
                isNegative = true;
            } else if (Character.isDigit(character)) {
                // If the current character is a digit, add it to the current number.
                number = number * 10 + character - '0';
            } else if (character == '[') {
                // If the current character is an open bracket, push an empty NestedInteger onto the stack.
                stack.push(new NestedInteger());
            } else if (character == ',' || character == ']') {
                // If the character is a comma or a close bracket,
                // and previous character was a digit, finalize and push the number onto the stack.
                if (Character.isDigit(s.charAt(i - 1))) {
                    if (isNegative) {
                        number = -number; // Apply the negative sign if applicable.
                    }
                    stack.peek().add(new NestedInteger(number)); // Add the number as a NestedInteger.
                }
                // Reset variables for processing the next number.
                number = 0;
                isNegative = false;

                // If the character is a close bracket and there is more than one NestedInteger on the stack,
                // pop the top NestedInteger and add it to the next NestedInteger on the stack.
                if (character == ']' && stack.size() > 1) {
                    NestedInteger topNestedInteger = stack.pop();
                    stack.peek().add(topNestedInteger);
                }
            }
        }

        // The top of the stack contains the deserialized NestedInteger.
        return stack.peek();
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    NestedInteger deserialize(string s) {
        // If the string does not start with '[', it means it is a single integer.
        if (s[0] != '[') {
            // Directly convert the string to an integer and return a NestedInteger object.
            return NestedInteger(stoi(s));
        }

        stack<NestedInteger> nestedStack; // Stack to maintain the level of nested lists.
        int num = 0; // To accumulate the number while traversing the string.
        bool isNegative = false; // Flag to check if the current number is negative.

        // Loop through each character of the string.
        for (int i = 0; i < s.size(); ++i) {
            if (s[i] == '-') {
                // If current character is '-', set the negative flag to true.
                isNegative = true;
            } else if (isdigit(s[i])) {
                // If the current character is a digit, add it to the current number.
                num = num * 10 + s[i] - '0';
            } else if (s[i] == '[') {
                // If the current character is '[', create a new NestedInteger and push it onto the stack.
                nestedStack.push(NestedInteger());
            } else if (s[i] == ',' || s[i] == ']') {
                // If the current character is ',' or ']', it may indicate the end of a number or a list.
                if (isdigit(s[i - 1])) {
                    // If the previous character was a digit, the current number is complete.
                    if (isNegative) {
                        // If the number is negative, negate it.
                        num = -num;
                    }
                    // Add the completed number to the top NestedInteger in the stack.
                    nestedStack.top().add(NestedInteger(num));
                }
                // Reset the number and the negative flag.
                num = 0;
                isNegative = false;
                if (s[i] == ']' && nestedStack.size() > 1) {
                    // If the current character is ']' and the stack size is greater than 1,
                    // it means we have completed a nested list.
                    NestedInteger topNestedInteger = nestedStack.top();
                    nestedStack.pop(); // Pop the top NestedInteger.
                    nestedStack.top().add(topNestedInteger); // Add the top NestedInteger to the new top of the stack.
                }
            }
        }

        // After processing the entire string, return the top of the stack which holds the deserialized result.
        return nestedStack.top();
    }
};

```

Typescript Solution

```typescript
// Function to deserialize a string into a NestedInteger.
// This can represent either a single integer or a nested list of integers.
function deserialize(s: string): NestedInteger {
    // If the string does not start with '[', it is an integer.
    if (s[0] !== '[') {
        return new NestedInteger(parseInt(s));
    }

    // Stack to hold the NestedIntegers as we build them.
    const stack: NestedInteger[] = [];
    let numberBuffer = 0; // Buffer to accumulate the digits of a number as we parse through the string.
    let isNegative = false; // Flag to mark the sign of the number currently being processed.

    // Iterate through each character in the string.
    for (let i = 0; i < s.length; ++i) {
        if (s[i] === '-') {
            // Found a minus sign, so the subsequent number is negative.
            isNegative = true;
        } else if (s[i] === '[') {
            // Start of a new NestedInteger (a nested list).
            stack.push(new NestedInteger());
        } else if (s[i] >= '0' && s[i] <= '9') {
            // Found a digit, add it to the buffer.
            numberBuffer = numberBuffer * 10 + parseInt(s[i]);
        } else if (s[i] === ',' || s[i] === ']') {
            // End of a number, or the end of the current list.
            if (s[i - 1] >= '0' && s[i - 1] <= '9') {
                // If the previous character was a number, add it to the most recent NestedInteger.
                stack[stack.length - 1].add(new NestedInteger(isNegative ? -numberBuffer : numberBuffer));
            }
            // Reset buffer and sign flag for the next number.
            numberBuffer = 0;
            isNegative = false;
            if (s[i] === ']' && stack.length > 1) {
                // End of a nested list, pop and add it to the next NestedInteger in the stack.
                const topNestedInteger = stack.pop();
                stack[stack.length - 1].add(topNestedInteger);
            }
        }
    }

    // The first element in the stack should be our fully parsed NestedInteger.
    return stack[0];
}

```

Time and Space Complexity

The time complexity of the code is `O(n)` where `n` is the length of the input string `s`. This is because the function involves a single loop through the input string, performing a constant amount of work for each character in the string.

The space complexity of the code is `O(n)`, also dependent on the length of the string `s`. In the worst case, the input string could represent a deeply nested list, requiring a new `NestedInteger` object for every `[` encountered before any `]` is encountered, which are stored in the stack `stk`. In the worst-case scenario, this stack could have as many nested `NestedInteger` objects as there are characters in the input string if the structure were to be very unbalanced. However, this is a very conservative estimation. In practical scenarios, the number of `NestedInteger` objects will often be less than `n`.