Documentation

The problem requires decoding a string encoded with the pattern k[encoded_string], where k is a positive integer indicating how many times the encoded_string inside the square brackets should be repeated. The input string is always valid, meaning there are no extra spaces, and the square brackets are well-formed. Additionally, the original string does not contain any digits except for the repetition counts. The goal is to return the fully decoded string. For example, the input "3[a]2[bc]" should be decoded to "aaabcbc".

The challenge lies in handling nested patterns, such as 3[a2[c]], which requires processing multiple levels of encoding. A stack-based approach is intuitive because it allows us to manage the current state (characters and counts) when encountering nested structures. By using a stack, we can save the current string and count whenever we encounter an opening bracket [and retrieve them when we encounter a closing bracket], ensuring that nested patterns are handled correctly.

Several key observations guide the solution. First, the string can contain nested patterns, meaning one encoded string can be inside another. For example, 3[a2[c]] requires decoding the inner 2[c] first. Second, digits represent the repetition count, while letters represent the characters to be repeated. Multi-digit numbers, such as 12[a], must be handled carefully. Third, a stack is ideal for managing nested structures because it allows us to save the current state and backtrack when a closing bracket] is encountered. Finally, the input string is always valid, so we do not need to handle edge cases like mismatched brackets or invalid characters.

The solution uses a stack to keep track of the current state (characters and counts) whenever a nested structure is encountered. It iterates through the string, parsing digits to handle multi-digit counts, pushing the current string and count onto the stack when encountering [, and popping from the stack when encountering] to repeat the current string by the count and append it to the popped string. Letters are appended directly to the current string. After processing the entire string, the decoded string is returned.

The solution must handle various edge cases, including single-level encoding (e.g., "3[a]" decoding to "aaa"), nested encoding (e.g., "3[a2[c]]" decoding to "accaccacc"), strings with no encoding (e.g., "abc" remaining unchanged), multiple encodings (e.g., "2[abc]3[cd]ef" decoding to "abcabccdcdcdef"), and large repetition counts

(e.g., "10[a]" decoding to "aaaaaaaaaa"). The time complexity of the solution is O(n), where (n) is the length of the input string, as each character is processed once. The space complexity is O(n) in the worst case, occurring when the stack grows to the size of the input string, such as in highly nested structures. Alternative approaches include a recursive method, which is less efficient in terms of space due to the recursion stack, and a two-pass approach, which involves building a tree-like structure and traversing it to construct the decoded string. However, the stack-based approach is the most efficient and intuitive.

Test cases validate the solution's correctness. For example, the input "3[a]2[bc]" decodes to "aaabcbc", as 3[a] becomes "aaa" and 2[bc] becomes "bcbc". Similarly, "3[a2[c]]" decodes to "accaccacc", as 2[c] becomes "cc", a2[c] becomes "acc", and 3[acc] becomes "accaccacc". Another example, "2[abc]3[cd]ef", decodes to "abcabccdcdcdef", as 2[abc] becomes "abcabc", 3[cd] becomes "cdcdcd", and ef remains unchanged.

The stack-based approach efficiently handles nested structures and ensures the decoding process is completed in linear time. The key insight is to use the stack to manage the state of the current string and count whenever a nested pattern is encountered. This problem is an excellent example of how stacks can simplify the handling of nested or recursive structures, making it a valuable exercise for understanding stack-based algorithms.