394. Decode String

Given an encoded string, return its decoded string.

The encoding rule is: `k[encoded_string]`, where the `encoded_string` inside the square brackets is being repeated exactly k times. Note that k is guaranteed to be a positive integer.

You may assume that the input string is always valid; there are no extra white spaces, square brackets are well-formed, etc. Furthermore, you may assume that the original data does not contain any digits and that digits are only for those repeat numbers, k. For example, there will not be input like `3a` or `2[4]`.

The test cases are generated so that the length of the output will never exceed 10^5.

Example 1:

Input: `s = "3[a]2[bc]"`
Output: `"aaabcbc"`

Example 2:

Input: `s = "3[a2[c]]"`
Output: `"accaccacc"`

Example 3:

Input: `s = "2[abc]3[cd]ef"`
Output: `"abcabccdcdcdef"`

Constraints:

- `1 <= s.length <= 30`
- `s` consists of lowercase English letters, digits, and square brackets `'[]'`.
- `s` is guaranteed to be a valid input.
- All the integers in `s` are in the range `[1, 300]`.

Solution

Solution

Here is the definition of an **encoded** string:

- a string is **encoded** if it only consists of lowercase English letters
- a string is **encoded** if it's in the form `k[s]` where k is a positive integer and `s` is a **encoded** string
- a string is **encoded** if it's a concatenation of two **encoded** strings

We can notice that an **encoded** string can be a concatenation of multiple **encoded** strings. To decode this string, we can separate the string into the multiple **encoded** strings, decode them separately and finally concatenate all the decoded strings together.

Here, we're solving a problem that can be broken down into the same repetitive problem. Thus, we can use recursion. The function `decodeString()` will return the decoded string of any **encoded** string.

There's two basic cases we should consider:

- The string is consists of only lowercase English letters. In this case, we can just return the original string.
- a string in the form `k[s]` where `s` is an **encoded** string and k is an integer. We can build the answer by concatenating k copies of `decodeString(s)`.

Any **encoded** string is just a concatenation of these cases. For any string, we'll first break it down into a bunch of strings that follow one of the two basic cases. Then we'll decode those separately and concatenate them together in the end.

For example, the string `5[abc3[ba]]jkl4[xyz]` can be separated into `5[abc3[ba]]`, `jkl`, and `4[xyz]`. We can find the decoded strings separately by using the function `decodeString()` and then we'll concatenate them together.

For the basic case in the form `k[s]`, how will we find the matching close bracket? When iterating through the string, we know we have found the correct close bracket when we reach a point in the string where the number of open brackets match the number of close brackets. In the same example, `5[abc3[ba` is currently incomplete since we only have 1 close bracket but 2 open brackets. `5[abc3[ba]]` however, is complete since we have 2 open and close brackets.

Time Complexity

Let L represent the length of the final string we return. Since we construct a string of length L, our time complexity is $\mathcal{O}(L)$.

Time Complexity: $\mathcal{O}(L)$

Space Complexity

Since we build a string with length L, our space complexity is also $\mathcal{O}(L)$.

Space Complexity: $\mathcal{O}(L)$

C++ Solution

```cpp
class Solution {
    int stringToInteger(string s) {
        int ans = 0;
        for (char nxt : s) {
            ans *= 10;
            ans += nxt - '0';
        }
        return ans;
    }

    public:
     string decodeString(string s) {
        string ans = "";
        int prev = 0;
        int repetitions = 0;
        int depth = 0;  // keeps track of # open bracket - # close bracket
        for (int i = 0; i < s.size(); i++) {
            if (depth == 0 && 'a' <= s[i] && s[i] <= 'z') {
                    // case with lowercase letters
                ans.push_back(s[i]);
                prev = i + 1;
            }
            if (s[i] == '[') {
                depth++;
                if (depth == 1) {  // open bracket for the case "k[s]"
                    repetitions = stringToInteger(s.substr(prev, i - prev));
                    prev = i + 1;
                }
            } else if (s[i] == ']') {
                depth--;
                if (depth == 0) {                   // close bracket for the case "k[s]"
                    while (repetitions > 0) {  // add k copies of s
                        ans += decodeString(s.substr(prev, i - prev));
                        repetitions--;
                    }
                    prev = i + 1;
                }
            }
        }
        return ans;
    }
};
```

Java Solution

```java
class Solution {
    private int stringToInteger(String s) {
        int ans = 0;
        for (int i = 0; i < s.length(); i++) {
            ans *= 10;
            ans += s.charAt(i) - '0';
        }
        return ans;
    }
    public String decodeString(String s) {
        StringBuilder ans = new StringBuilder();
        int prev = 0;
        int repetitions = 0;
        int depth = 0; // keeps track of # open bracket - # close bracket
        for (int i = 0; i < s.length(); i++) {
            if (depth == 0 && 'a' <= s.charAt(i) && s.charAt(i) <= 'z') {
                    // case with lowercase letters
                ans.append(s.charAt(i));
                prev = i + 1;
            }
            if (s.charAt(i) == '[') {
                depth++;
                if (depth == 1) { // open bracket for the case "k[s]"
                    repetitions = stringToInteger(s.substring(prev, i));
                    prev = i + 1;
                }
            } else if (s.charAt(i) == ']') {
                depth--;
                if (depth == 0) { // close bracket for the case "k[s]"
                    while (repetitions > 0) { // add k copies of s
                        ans.append(decodeString(s.substring(prev, i)));
                        repetitions--;
                    }
                    prev = i + 1;
                }
            }
        }
        return ans.toString();
    }
}
```

Python Solution

Note: The same recursion here will be done with the function `decode()`.

```python
class Solution:
    def decodeString(self, s: str) -> str:
        def stringToInteger(s):
            ans = 0
            for ch in s:
                ans *= 10
                ans += int(ch) - int("0")
            return ans

        def decode(s):
            ans = str()
            prev = 0
            repetitions = 0
            depth = 0  # keeps track of # open bracket - # close bracket
            for i in range(len(s)):
                if (depth == 0 and "a" <= s[i] and s[i] <= "z"):
                    # case with lowercase letters
                    ans += s[i]
                    prev = i + 1
                if s[i] == "[":
                    depth += 1
                    if depth == 1:  # open bracket for the case "k[s]"
                        repetitions = stringToInteger(s[prev:i])
                        prev = i + 1
                elif s[i] == "]":
                    depth -= 1
                    if depth == 0:  # close bracket for the case "k[s]"
                        while repetitions > 0:  # add k copies of s
                            ans += decode(s[prev:i])
                            repetitions -= 1
                        prev = i + 1
            return ans

        return decode(s)
```