

394. Decode String

[Leetcode Link](#)

Given an encoded string, return its decoded string.

The encoding rule is: $k[\text{encoded_string}]$, 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 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

```
1 class Solution {
2     int stringToInteger(string s) {
3         int ans = 0;
4         for (char nxt : s) {
5             ans *= 10;
6             ans += nxt - '0';
7         }
8         return ans;
9     }
10
11 public:
12     string decodeString(string s) {
13         string ans = "";
14         int prev = 0;
15         int repetitions = 0;
16         int depth = 0; // keeps track of # open bracket - # close bracket
17         for (int i = 0; i < s.size(); i++) {
18             if (depth == 0 && 'a' <= s[i] && s[i] <= 'z') {
19                 // case with lowercase letters
20                 ans.push_back(s[i]);
21                 prev = i + 1;
22             }
23             if (s[i] == '[') {
24                 depth++;
25                 if (depth == 1) { // open bracket for the case "k[s]"
26                     repetitions = stringToInteger(s.substr(prev, i - prev));
27                     prev = i + 1;
28                 }
29             } else if (s[i] == ']') {
30                 depth--;
31                 if (depth == 0) { // close bracket for the case "k[s]"
32                     while (repetitions > 0) { // add k copies of s
33                         ans += decodeString(s.substr(prev, i - prev));
34                         repetitions--;
35                     }
36                     prev = i + 1;
37                 }
38             }
39         }
40         return ans;
41     }
42 };
```

Java Solution

```
1 class Solution {
2     private int stringToInteger(String s) {
3         int ans = 0;
4         for (int i = 0; i < s.length(); i++) {
5             ans *= 10;
6             ans += s.charAt(i) - '0';
7         }
8         return ans;
9     }
10
11 public String decodeString(String s) {
12     StringBuilder ans = new StringBuilder();
13     int prev = 0;
14     int repetitions = 0;
15     int depth = 0; // keeps track of # open bracket - # close bracket
16     for (int i = 0; i < s.length(); i++) {
17         if (depth == 0 && 'a' <= s.charAt(i) && s.charAt(i) <= 'z') {
18             // case with lowercase letters
19             ans.append(s.charAt(i));
20             prev = i + 1;
21         }
22         if (s.charAt(i) == '[') {
23             depth++;
24             if (depth == 1) { // open bracket for the case "k[s]"
25                 repetitions = stringToInteger(s.substring(prev, i));
26                 prev = i + 1;
27             }
28         } else if (s.charAt(i) == ']') {
29             depth--;
30             if (depth == 0) { // close bracket for the case "k[s]"
31                 while (repetitions > 0) { // add k copies of s
32                     ans.append(decodeString(s.substring(prev, i)));
33                     repetitions--;
34                 }
35                 prev = i + 1;
36             }
37         }
38     }
39     return ans.toString();
40 }
```

Python Solution

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

```
1 class Solution:
2     def decodeString(self, s: str) -> str:
3         def stringToInteger(s):
4             ans = 0
5             for ch in s:
6                 ans *= 10
7                 ans += int(ch) - int("0")
8             return ans
9
10        def decode(s):
11            ans = str()
12            prev = 0
13            repetitions = 0
14            depth = 0 # keeps track of # open bracket - # close bracket
15            for i in range(len(s)):
16                if (depth == 0 and "a" <= s[i] and s[i] <= "z"):
17                    # case with lowercase letters
18                    ans += s[i]
19                    prev = i + 1
20                if s[i] == "[":
21                    depth += 1
22                    if depth == 1: # open bracket for the case "k[s]"
23                        repetitions = stringToInteger(s[prev:i])
24                        prev = i + 1
25                elif s[i] == "]":
26                    depth -= 1
27                    if depth == 0: # close bracket for the case "k[s]"
28                        while repetitions > 0: # add k copies of s
29                            ans += decode(s[prev:i])
30                            repetitions -= 1
31                            prev = i + 1
32            return ans
33
34        return decode(s)
```

Got a question? [Ask the Teaching Assistant](#) anything you don't understand.