# Question

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; 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 won't be input like 3a or 2[4].

**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"

**Example 4:**

**Input:** s = "abc3[cd]xyz"

**Output:** "abccdcdcdxyz"

**Constraints:**

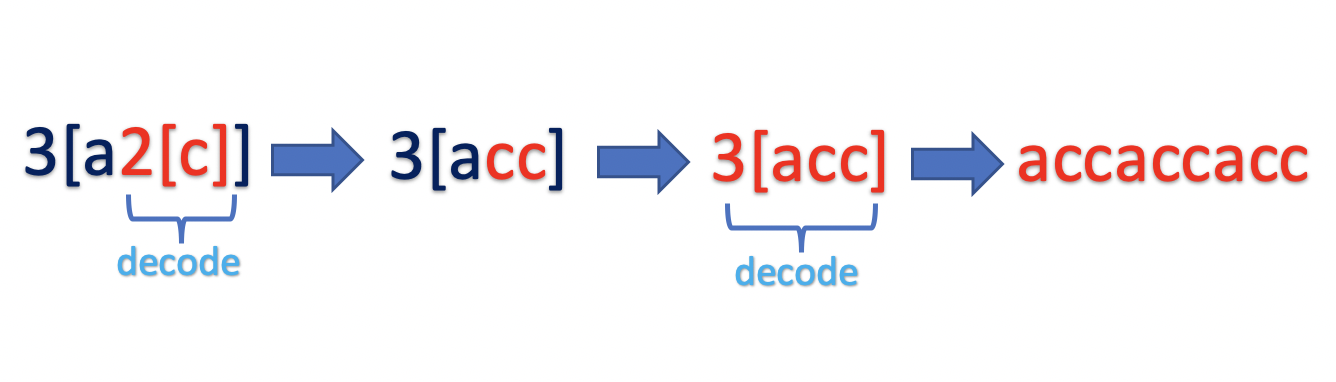
* 1 <= s.length <= 30
* s consists of lowercase English letters, digits, and square brackets '[]'.
* s is guaranteed to be **a valid** input.
* All the integers in s are in the range [1, 300].

# Solution

#### **Overview**

We are given string s*s* in a particular form k[string] and we have to decode it as string repeated k times . For example,2[b] is decoded as bb.

The problem seems straightforward at first glance. But the trick here is that there can be nested encoded strings like k[string k[string]]. For example, string s =3[a2[c]]. In such cases, we must decode the innermost string and continue in an outward direction until the entire string is decoded.



If you have solved similar problem such as [Evaluate Polish Notation](https://leetcode.com/problems/evaluate-reverse-polish-notation/) or [Simplify Path](https://leetcode.com/problems/simplify-path/) , it is clear that [Stack Data Structure](https://en.wikipedia.org/wiki/Stack_(abstract_data_type)) is best suited to implement such problems. We could implement a stack data structure or recursively build the solution by using an internal call stack. Let's understand both approaches in detail.

#### **Approach 1: Using Stack**

**Intuition**

We have to decode the result in a particular pattern. We know that the input is always valid. The pattern begins with a number k, followed by opening braces [, followed by string. Post that, there could be one of the following cases :

1. There is another nested pattern k[string k[string]]
2. There is a closing bracket k[string]

Since we have to start decoding the innermost pattern first, continue iterating over the string s, pushing each character to the stack until it is not a closing bracket ]. Once we encounter the closing bracket ], we must start decoding the pattern.

As we know that stack follows the Last In First Out (LIFO) Principle, the top of the stack would have the data we must decode.

**Algorithm**

The input can contain an alphabet (a-z), digit (0-9), opening braces [ or closing braces ]. Start traversing string s and process each character based on the following rules:

Case 1) Current character is not a closing bracket ].

Push the current character to stack.

Case 2) Current character is a closing bracket ].

Start decoding the last traversed string by popping the string decodedString and number k from the top of the stack.

* Pop from the stack while the next character is not an opening bracket [ and append each character (a-z) to the decodedString.
* Pop opening bracket [ from the stack.
* Pop from the stack while the next character is a digit (0-9) and build the number k.

Now that we have k and decodedString , decode the pattern k[decodedString] by pushing the decodedString to stack k times.

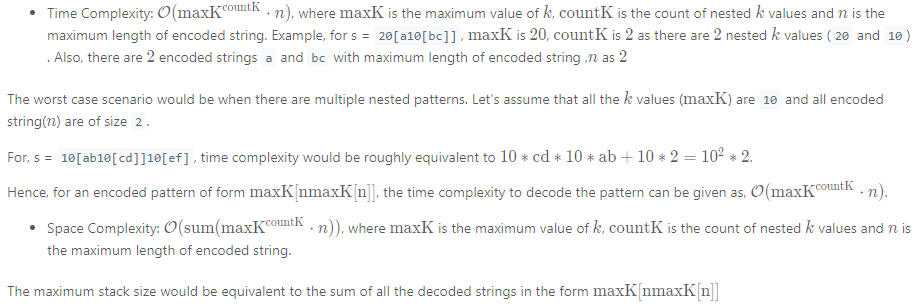
Once the entire string is traversed, pop the result from stack and return.

1 / 17

**Implementation**

|  |
| --- |
| class Solution {  public String decodeString(String s) {  Stack<Character> stack = new Stack<>();  for (int i = 0; i < s.length(); i++) {  if (s.charAt(i) == ']') {  List<Character> decodedString = new ArrayList<>();  // get the encoded string  while (stack.peek() != '[') {  decodedString.add(stack.pop());  }  // pop [ from the stack  stack.pop();  int base = 1;  int k = 0;  // get the number k  while (!stack.isEmpty() && Character.isDigit(stack.peek())) {  k = k + (stack.pop() - '0') \* base;  base \*= 10;  }  // decode k[decodedString], by pushing decodedString k times into stack  while (k != 0) {  for (int j = decodedString.size() - 1; j >= 0; j--) {  stack.push(decodedString.get(j));  }  k--;  }  }  // push the current character to stack  else {  stack.push(s.charAt(i));  }  }  // get the result from stack  char[] result = new char[stack.size()];  for (int i = result.length - 1; i >= 0; i--) {  result[i] = stack.pop();  }  return new String(result);  }  } |

**Complexity Analysis**



#### **Approach 2: Using 2 Stack**

**Intuition**

In the previous approach, we used a single character stack to store the digits(0-9) as well as letters (a-z). We could instead maintain 2 separate stacks.

* countStack: The stack would store all the integer k.
* stringStack: The stack would store all the decoded strings.

Also, instead of pushing the decoded string to the stack character by character, we could improve our algorithm by appending all the characters into the string first and then push the entire string into the stringStack. Let's look at the algorithm in detail.

**Algorithm**

Iterate over the string s and process each character as follows:

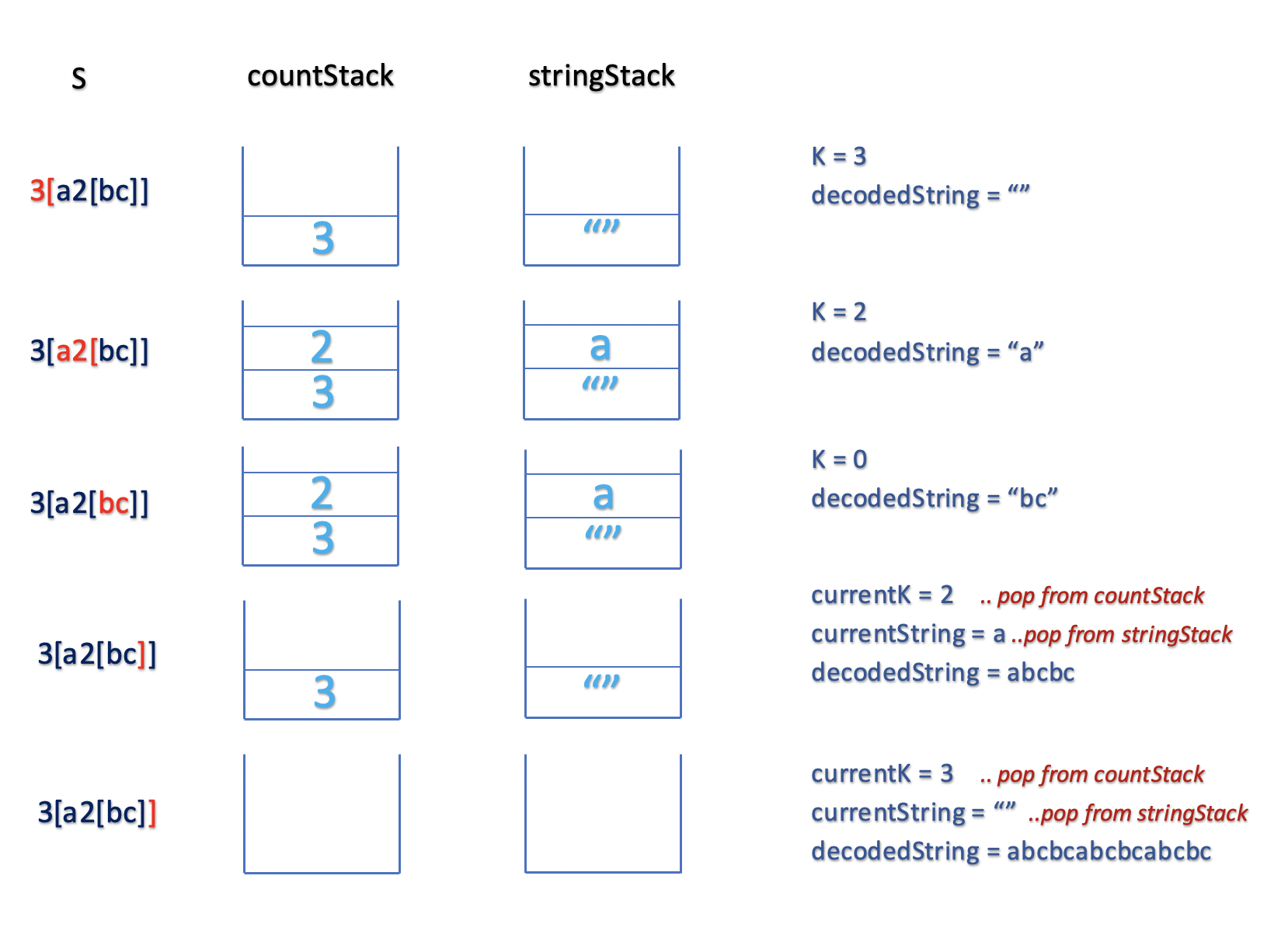
Case 1) If the current character is a digit (0-9), append it to the number k.

Case 2) If the current character is a letter (a-z), append it to the currentString.

Case 3) If current character is a opening bracket [, push k and currentString intocountStack and stringStack respectively.

Case 4) Closing bracket ]: We must begin the decoding process,

* We must decode the currentString. Pop currentK from the countStack and decode the pattern currentK[currentString]
* As the stringStack contains the previously decoded string, pop the decodedString from the stringStack. Update the decodedString = decodedString + currentK[currentString]



**Implementation**

|  |
| --- |
| class Solution {  String decodeString(String s) {  Stack<Integer> countStack = new Stack<>();  Stack<StringBuilder> stringStack = new Stack<>();  StringBuilder currentString = new StringBuilder();  int k = 0;  for (char ch : s.toCharArray()) {  if (Character.isDigit(ch)) {  k = k \* 10 + ch - '0';  } else if (ch == '[') {  // push the number k to countStack  countStack.push(k);  // push the currentString to stringStack  stringStack.push(currentString);  // reset currentString and k  currentString = new StringBuilder();  k = 0;  } else if (ch == ']') {  StringBuilder decodedString = stringStack.pop();  // decode currentK[currentString] by appending currentString k times  for (int currentK = countStack.pop(); currentK > 0; currentK--) {  decodedString.append(currentString);  }  currentString = decodedString;  } else {  currentString.append(ch);  }  }  return currentString.toString();  }  } |

**Complexity Analysis**

Assume, n*n* is the length of the string s*s*.

* Time Complexity: O(maxK⋅*n*), where maxK  is the maximum value of k*k* and n*n* is the length of a given string s*s*. We traverse a string of size n*n* and iterate k*k* times to decode each pattern of form \text{k[string]}k[string]. This gives us worst case time complexity as \mathcal{O}(\text{maxK} \cdot n)O(maxK⋅*n*).
* Space Complexity: \mathcal{O}(m+n)O(*m*+*n*), where m*m* is the number of letters(a-z) and n*n* is the number of digits(0-9) in string s*s*. In worst case, the maximum size of \text{stringStack}stringStack and \text{countStack}countStack could be m*m* and n*n* respectively.

#### **Approach 3: Using Recursion**

**Intuition**

In the previous approach, we implemented an external stack to keep the track of each character traversed. Ideally, a stack is required when we have nested encoded string in the form k[string k[string]].

Using this intuition, we could start by building k and string and recursively decode for each nested substring. The recursion uses an internal call stack to store the previous state. Let's understand the algorithm in detail.

**Algorithm**

* Build result while next character is letter (a-z) and build the number k while next character is a digit (0-9) by iterating over string s.
* Ignore the next [ character and recursively find the nested decodedString.
* Decode the current pattern k[decodedString] and append it to the result.
* Return the current result.

The above steps are repeated recursively for each pattern until the entire string s is traversed.

Base Condition: We must define a base condition that must be satisfied to backtrack from the recursive call. In this case, we would backtrack and return the result when we have traversed the string s or the next character is ] and there is no nested substring.

Thanks to [@bluedawnstar](https://leetcode.com/bluedawnstar/) for suggesting the solution.

**Implementation**

|  |
| --- |
| class Solution {  int index = 0;  String decodeString(String s) {  StringBuilder result = new StringBuilder();  while (index < s.length() && s.charAt(index) != ']') {  if (!Character.isDigit(s.charAt(index)))  result.append(s.charAt(index++));  else {  int k = 0;  // build k while next character is a digit  while (index < s.length() && Character.isDigit(s.charAt(index)))  k = k \* 10 + s.charAt(index++) - '0';  // ignore the opening bracket '['  index++;  String decodedString = decodeString(s);  // ignore the closing bracket ']'  index++;  // build k[decodedString] and append to the result  while (k-- > 0)  result.append(decodedString);  }  }  return new String(result);  }  } |

**Complexity Analysis**

Assume, n*n* is the length of the string s*s*.

* Time Complexity: O(maxK⋅*n*) as in Approach 2
* Space Complexity: \mathcal{O}(n)O(*n*). This is the space used to store the internal call stack used for recursion. As we are recursively decoding each nested pattern, the maximum depth of recursive call stack would not be more than n*n*