# 880. Decoded String at Index

Medium Topics String

## Problem Description

You are tasked with deciphering an encoded string `s`. The encoding process operates under two simple rules. If you encounter a letter during the decoding process, you simply write it down. However, if you run into a digit `d`, then you need to replicate the entire decoded string (up until that point), `d - 1` additional times.

Now, with this encoded string, you're given a task to find out the `kth` letter in the resultant decoded string where `k` is 1-indexed. The challenge lies in the fact that the decoded string can potentially become very large, and thus, it may not be feasible to decode the entire string just to find one character.

## Intuition

The solution approach is built upon a key observation to avoid decoding the entire string, which could consume a lot of time and space, particularly if the string is very large.

To begin with, we figure out the full length of the decoded string without actually decoding it. This is possible by iterating over the encoded string and applying the rules: we increment our length by 1 for a letter, and we multiply our current length by `d` when we encounter a digit.

Once we have the total length, we then think in reverse: starting from the end of the encoded string, we work backwards given that the structure of the encoded string is inherently repetitive and redundant due to the digits. By doing modulus `k % m` at each step (where `m` is the length at that point), we are effectively peeling off the outer layers of this repetitive structure, reducing `k` to find the position in the smaller version of the string.

When we come across a digit while iterating backwards, it tells us that the current string is a repetition caused by this digit, so we divide `m` by the digit to step into the inner layer of the next repetition, undoing the previous expansion. If the character is a letter and `k` becomes 0, it means we have successfully reversed into the original non-repeated string with the correct position, and we can thus return this character.

The trick here is recognizing that instead of expanding the encoded string, we shrink `k` until we can directly map it to the correct character. By reversing the process (both the string and the decoding steps), we significantly optimize for both time and space.

## Solution Approach

The implementation follows a two-pass approach:

In the first pass, we traverse the encoded string `s` from start to end to calculate the total length `n` of the decoded string. This is done by iterating through each character in the string:

- If the character is a letter, we increment `n` by 1, because each letter adds a single character to the decoded string.
- If the character is a digit `d`, we multiply `n` by `d`. This is because the character indicates that the sequence before this digit should be repeated `d` times.

After the first pass, we have the total length `n` without having to store the decoded string.

In the second pass, we reverse iterate the string (hence the use of `s[i::-1]`) and decrement `k` modulo `n`. The modulo operation effectively reduces the size of `k` to fit within the current size of the non-repeated string.

During this reverse iteration:

- Each time we encounter a digit `d`, it indicates the start of a multiplied segment. Therefore, we divide `n` by `d` to step backwards out of the current repeated segment.
- When we encounter a letter, we subtract 1 from `n`. If at this point `k` equals 0 and the current character is a letter, this means we have found the `kth` character in the decoded string.

The use of modulo operation (`k %= n`) is particularly crucial as it keeps shrinking `k` to stay within the bounds of the "virtual" decoded string we are constructing.

This approach avoids the need to construct the entire decoded string, which is critical for memory efficiency, especially when the size of the decoded string is enormous. It is an excellent example of space optimization, where the understanding of the problem's constraints (such as the predictable repetitiveness of the encoded string) is leveraged to eliminate unnecessary work.

Discover Your Strengths and Weaknesses: Take Our 2-Minute Quiz to Tailor Your Study Plan:

### Example Walkthrough

Let's go through an example to illustrate the solution approach.

Consider the encoded string `s = "2[abc]3[cd]ef"`, and we want to determine the 9th letter in the decoded string.

In the first pass, we would calculate the full length `n` of the decoded string as follows:

- We start with `n = 0` and we encounter `2`, so we skip it since we have not encountered any letters yet.
- We encounter letters `abc` and increment `n` by 3 (for each letter), so `n = 3`.
- The next digit is `3`, which tells us to multiply the current `n` by `3`, so `n = 3 * 2 = 6`.
- We then encounter `3[cd]`. After `cd` our `n` becomes `n = 6 + 2 = 8` and then seeing `3` we multiply `n` by `3`, so `n = 8 * 3 = 24`.
- Finally, we have the letters `ef` which add `2` more to `n`, making it `24 + 2 = 26`.

So, without decoding, we know the length `n` of the decoded string would be `26`.

In the second pass, we work backwards to find the 9th letter:

- We start with `k = 9`. Going in reverse, we skip the first two letters `fe` since `k` is greater than `2`.
- We then see digit `3`. We divide our total length `n` by `3`, giving us `n = 26 / 3 = 8`. Here `k` is still greater than `6`, so we continue.
- We encounter `dc`. Since `k` is `8` and we are looking for one character, we do `k %= n`, which doesn't change `k` since `8 < 8`.
- The next digit is `2`, we divide `n` by `2` to get `n = 8 / 2 = 4`. Now, `k = 8` is greater than `4`, so we do `k %= n` and get `k = 1`.
- We will now encounter the letters `cba` in reverse. Since `k = 1`, once we decrement `n` by `1` three times (while skipping over letters `cba`), we reach `k = 0` on the letter `a`.

Once `k` becomes `0`, the current character, `a`, is the 9th letter in the decoded string. Note that we did not need to decode the entire string, which saves a significant amount of time and space. This strategy is particularly effective as the strings grow in length and complexity.

## Solution Implementation

### Python

```python
class Solution:
    def decodeAtIndex(self, tape: str, k: int) -> str:
        size = 0  # This will represent the size of the decoded string

        # Calculate size of the decoded string
        for char in tape:
            if char.isdigit():
                size *= int(char)  # If character is a digit, multiply the size
            else:
                size += 1  # If character is a letter, increment the size

        # Iterate over the string in reverse to find the k-th character
        for char in reversed(tape):
            k %= size  # k could be larger than size, ensure we cycle within bounds of 'size'
            if k == 0 and char.isalpha():
                return char  # If k is 0 and char is a letter, this is the answer

            # If character is a digit, divide the size,
            # else if character is a letter, decrease size
            if char.isdigit():
                size //= int(char)  # Divide the size by the digit
            else:
                size -= 1  # Decrease the size for the letter

# Sample usage:
sol = Solution()
result = sol.decodeAtIndex("leet2code3", 10)
print(result)  # Should print the 10th character in the decoded string
```

### Java

```java
class Solution {

    // Decodes a string at the specified index 'k'
    public String decodeAtIndex(String S, int K) {
        long size = 0; // This will hold the size of the decoded string until a given point

        // First pass: Calculate the size of the decoded string
        for (int i = 0; i < S.length(); ++i) {
            char c = S.charAt(i);
            if (Character.isDigit(c)) {
                // If it's a digit, multiply the current size by the digit to simulate the repeat-expansion
                size *= c - '0';
            } else {
                // If it's a letter, increment the size
                ++size;
            }
        }

        // Second pass: Work backwards to find the k-th character
        for (int i = S.length() - 1; ; --i) {
            K %= size; // Modulate K with current size to handle wrap around cases
            char c = S.charAt(i);

            if (K == 0 && (Character.isDigit(c)) {
                // If K is zero, and we're at a character, that's the character we want to return
                return String.valueOf(c);
            }

            if (Character.isDigit(c)) {
                // If it's a digit, we reverse the expansion by dividing the size
                size /= c - '0';
            } else {
                // If it's an alphabet character, decrement the size as we move left
                --size;
            }
        }
    }
}
```

### C++

```cpp
#include <string>
#include <cctype> // for isdigit and isalpha functions

class Solution {
public:
    /**
     * Decodes the string at the specified index.
     *
     * Each letter (lowercase and uppercase) in the string will be written to a tape.
     * For each number in the string, the entire current tape is repeated that number of times.
     * Given a string that has been encoded in this way and an index k, return the k-th letter (1-indexed).
     *
     * @param s The encoded string consisting of lowercase letters, uppercase letters, and digits.
     * @param k The 1-indexed position in the decoded string.
     * @return A single character string representing the k-th letter in the decoded string.
     */
    string decodeAtIndex(string s, int k) {
        // Using a long long to handle the possible length of the decoded string.
        long long decodedLength = 0;

        // First pass to calculate the length of the decoded string.
        for (const char& c : s) {
            if (isdigit(c)) {
                decodedLength *= (c - '0'); // Multiplies the current length by the digit.
            } else {
                ++decodedLength; // Increment the length for a character.
            }
        }

        // Reverse iteration over the encoded string.
        for (int i = s.size() - 1; ; --i) {
            // Modulo operation to find the effective index.
            k %= decodedLength;
            // If k is 0 and character is an alphabet, return this character.
            if (k == 0 && isalpha(s[i])) {
                return string(1, s[i]);
            }

            // If current character is a digit, divide decodedLength by the digit.
            // This effectively reverses the encoding process.
            if (isdigit(s[i])) {
                decodedLength /= (s[i] - '0');
            } else {
                // If it's an alphabet decrement the length count.
                --decodedLength;
            }
        }

        // Given the problem constraints, we should never reach this point.
        // This is here to satisfy the compiler warning of reaching end of non-void function.
        return "";
    }
};
```

### TypeScript

```typescript
function decodeAtIndex(S: string, K: number): string {
    // `decodedSize` represents the length of the decoded string.
    let decodedSize = 0n;

    // Calculate the length of the fully expanded string without actual expansion.
    for (const char of S) {
        if (char >= '2' && char <= '9') {
            decodedSize *= BigInt(char);
        } else {
            ++decodedSize;
        }
    }

    // Starting from the end, try to find the character at the K-th position.
    for (let i = S.length - 1; ; --i) {
        // Adjust K to be within the bounds of the current pattern.
        K %= Number(decodedSize);

        // If K is 0 or matches the current character, return this character.
        if (K === 0 && S[i] >= 'a' && S[i] <= 'z') {
            return S[i];
        }

        // If the character is a digit, shrink the decoded size accordingly.
        if (S[i] >= '2' && S[i] <= '9') {
            decodedSize = BigInt(S[i]);
        } else {
            // If the character is a letter, reduce the size as we move past the character.
            --decodedSize;
        }
    }

    // The function should never reach this point because the loops should return a value.
    // However, TypeScript requires a return at the end, so we throw an error to indicate abnormal behaviour.
    throw new Error('No character found at the given index');
}
```

```
class Solution:
    def decodeAtIndex(self, tape: str, k: int) -> str:
        size = 0  # This will represent the size of the decoded string

        # Calculate size of the decoded string
        for char in tape:
            if char.isdigit():
                size *= int(char)  # If character is a digit, multiply the size
            else:
                size += 1  # If character is a letter, increment the size

        # Iterate over the string in reverse to find the k-th character
        for char in reversed(tape):
            k %= size  # k could be larger than size, ensure we cycle within bounds of 'size'
            if k == 0 and char.isalpha():
                return char  # If k is 0 and char is a letter, this is the answer

            # If character is a digit, divide the size,
            # else if character is a letter, decrease size
            if char.isdigit():
                size //= int(char)  # Divide the size by the digit
            else:
                size -= 1  # Decrease the size for the letter

# Sample usage:
sol = Solution()
result = sol.decodeAtIndex("leet2code3", 10)
print(result)  # Should print the 10th character in the decoded string
```

## Time and Space Complexity

### Time Complexity

The time complexity of the provided code is $O(N)$, where $N$ is the length of the input string `s`. The reasoning behind this is as follows:

1. The first `for` loop iterates over each character of the string once, giving a complexity of $O(N)$ for this part.
2. The second `for` loop also iterates over each character of the string once, but in reverse order, which does not change the complexity, thus it remains $O(N)$ for this part.

Combining both loops that sequentially iterate over the string without any nested iterations leads to an overall time complexity of $O(N)$.

### Space Complexity

The space complexity of the code is $O(1)$. This is because:

1. A fixed number of single-value variables are used (`n` and `i`).
2. No additional data structures that grow with the input size are utilized.
3. The `for` loop utilizes reverse iteration (`s[i::-1]`) which in Python does create a new reversed string, but since the string is not stored and is only used for iteration, the memory remains constant.

In conclusion, the space complexity is unaffected by the size of the input string, and additional memory usage does not scale with $N$, hence $O(1)$.