271. Encode and Decode Strings

Medium Design Array String Leetcode Link

Problem Description

In this problem, we need to develop a method to encode and decode lists of strings so that they can be sent correctly over a network. The main challenge is to create a format for the encoded string that allows us to uniquely decipher each string in the list upon decoding without any ambiguity or loss of information. The encoded string must carry enough information to restore the original list of strings exactly.

Intuition

The intuition behind the solution is to prepend the length of each string to it before appending it to the encoded string, separating the length of the string from the string itself with a delimiter or using a fixed-width representation of the length. Upon decoding, we leverage the length information to extract each string accurately.

The chosen approach in the provided solution uses a fixed-width 4-character representation to store the length of each string. This approach simplifies the encoding and decoding process as follows:

- **Encoding**: For each string in the input list, we determine its length and format it to a 4-character string with padding, if necessary. This length prefix is then concatenated with the actual string. Once all strings have been processed this way, they are joined together to form the final encoded string.

- **Decoding**: To decode, we iterate over the encoded string, reading 4 characters at a time to determine the length of the next string. This length is converted to an integer, which is then used to extract the next string of that length from the encoded string. This process continues until the entire encoded string has been successfully decoded into the original list of strings.

The solution is clean and efficient as it avoids ambiguity (since the length prefix is fixed-width, there is no confusion about where each string starts and ends) and eschews the need for escape characters or complex parsing logic.

Solution Approach

The implementation of the solution uses basic string manipulation and list operations to achieve the desired result. The codec class created consists of two methods: `encode` and `decode`.

Encode Method

The `encode` method takes a list of strings as input. For each string in the input list, it performs the following steps:

1. It calculates the length of the string.
2. It then formats this length into a 4-character wide string, using Python's `.format()` method. This is done by the expression `'{:4}'.format(len(s))`, which ensures that the length is padded with spaces if it is less than 4 characters long. The use of fixed-width for the length ensures the encoded string can be correctly parsed during decoding.
3. The 4-character length prefix is concatenated with the actual string.
4. These resulting strings are appended to an accumulator list `ans`.

After processing all the strings, the `ans` list is joined into a single string without any delimiter between them and returned. This works because the fixed-width length prefix allows us to know exactly where each string begins and ends.

Decode Method

The `decode` method is responsible for reversing the encoding process. It takes the encoded single string as input and outputs the original list of strings. The process is as follows:

1. Initialize an empty list `ans` to store the decoded strings, and set `i = 0` to keep track of the current index in the encoded string, `s`.
2. While `i` is less than the length of `s`, perform the following steps in a loop: a. Read 4 characters from `i` to `i + 4` to get the string's length. Convert it to an integer with `int(s[i : i + 4])`. Since we know the length of each string is exactly 4 characters, we can directly slice out the length information. b. Update `i` to skip past the length prefix. c. Use the length to determine the substring that constitutes our original string, found at `s[i : i + size]`. d. Append this substring to the `ans` list. e. Update `i` to move past the current string, preparing to read the length of the next string.

Once the loop is finished (and thus, the entire string `s` has been parsed), the `ans` list contains all the original strings in the correct order. The list `ans` is returned to provide the decoded list of strings.

These methods form an efficient and robust solution for the problem, making use of simple data structures (strings and lists) and straightforward algorithmic patterns (iteration and substring extraction based on a fixed format).

Example Walkthrough

Let's assume we have the following list of strings that we want to encode and then decode:

`["hello", "world", "leetcode", "example"]`

The encoding process for each string in this example would be as follows:

1. Take the first string "hello":
   - Calculate the length: `5`
   - Format the length to 4-characters: `'   5'`
   - Concatenate the length and the string: `'   5hello'`
2. Now for the second string "world":
   - Calculate the length: `5`
   - Format the length to 4-characters: `'   5'`
   - Concatenate the length and the string: `'   5world'`
3. Follow the same steps for "leetcode":
   - Calculate the length: `8`
   - Format the length to 4-characters: `'   8'`
   - Concatenate the length and the string: `'   8leetcode'`
4. And finally for "example":
   - Calculate the length: `7`
   - Format the length to 4-characters: `'   7'`
   - Concatenate the length and the string: `'   7example'`

After encoding all elements of the list, we join them together to form the final encoded string without any delimiter:

Encoded String: `'   5hello   5world   8leetcode   7example'`

Now, for the decoding process, we start with the encoded string and decode it back into the original list of strings:

1. Initialize an empty list `ans` and set `i = 0`
2. While `i < len(s)`: (where `s` is the encoded string) a. Read 4 characters to get the length: `int('   5')` → `5` b. Update `i` by 4: `i = i + 4` c. Extract 5 characters starting from the new `i`: `"hello"` d. Append "hello" to `ans` e. Update `i` by 5 to move past the current string

Repeat steps a-e to decode the next strings:

- For "world", extract 5 characters after reading the length → `ans` becomes `["hello", "world"]`
- For "leetcode", extract 8 characters after reading the length → `ans` becomes `["hello", "world", "leetcode"]`
- For "example", extract 7 characters after reading the length → `ans` becomes `["hello", "world", "leetcode", "example"]`

Once we reach the end of the encoded string, the decoding process is complete. The final decoded list of strings is:

Decoded List: `["hello", "world", "leetcode", "example"]`

This walk-through demonstrates that our encoding scheme correctly maintains the integrity and order of the original list of strings when it is decoded back from the encoded string.

Python Solution

```python
class Codec:
    def encode(self, strs: List[str]) -> str:
        """
        Encodes a list of strings to a single string.

        Each string is prefixed with its length in a 4-character wide field,
        allowing for easy extraction during decoding.
        """
        encoded_string = []
        for string in strs:
            # '{:4}' formats the length into a 4-character wide field,
            # padding with spaces if the number is less than 4 characters long.
            length_prefix = '{:4}'.format(len(string))
            encoded_string.append(length_prefix + string)
        return ''.join(encoded_string)

    def decode(self, s: str) -> List[str]:
        """
        Decodes a single string to a list of strings.

        Each string was encoded with a 4-character length prefix, which we use to
        determine where one string ends and the next begins.
        """
        decoded_strings = []
        i = 0
        n = len(s)
        while i < n:
            # Extract the length of the next string, which is stored in the first 4 characters.
            size = int(s[i: i + 4])
            i += 4
            # Extract the string of the given length.
            decoded_strings.append(s[i: i + size])
            i += size
        return decoded_strings


# Example of how the Codec class is expected to be used:
# codec = Codec()
# encoded_data = codec.encode(strs)
# decoded_data = codec.decode(encoded_data)

```

Java Solution

```java
import java.util.List;
import java.util.ArrayList;

public class Codec {

    /**
     * Encodes a list of strings to a single string.
     *
     * @param strs the list of strings to encode
     * @return encoded single string
     */
    public String encode(List<String> strs) {
        // Use StringBuilder to efficiently build the encoded string
        StringBuilder encodedString = new StringBuilder();

        // Append the length of each string followed by the string itself to the builder
        for (String str : strs) {
            // Cast length to char to compactly store the length (only safe for strings of length 0-65535)
            encodedString.append((char) str.length()).append(str);
        }

        // Convert the StringBuilder to a String and return
        return encodedString.toString();
    }

    /**
     * Decodes a single string to a list of strings.
     *
     * @param s the encoded single string
     * @return decoded list of strings
     */
    public List<String> decode(String s) {
        // Create a list to hold the decoded strings
        List<String> decodedStrings = new ArrayList<>();

        // Initialize an index to iterate through the encoded string
        int index = 0;
        int length = s.length();

        // Iterate through the encoded string and decode the strings
        while (index < length) {
            // Read the size of the next string
            int size = s.charAt(index++);
            // Extract the actual string by its size and add it to the collection
            decodedStrings.add(s.substring(index, index + size));
            // Move the index past the retrieved string
            index += size;
        }

        // Return the list of decoded strings
        return decodedStrings;
    }
}

/* Usage example:
// Instantiate the codec
Codec codec = new Codec();
// Encode and decode
List<String> strs = codec.decode(codec.encode(strs));
*/

```

C++ Solution

```cpp
#include <string>
#include <vector>

class Codec {
public:
    // Encodes a list of strings to a single string.
    // Each string's length is stored as a fixed-size prefix before the actual string content.
    string encode(const vector<string>& strs) {
        string encodedString;

        for (const string& str : strs) {
            // Convert the size to a string of bytes and append it to the result.
            int size = str.size();
            encodedString.append(reinterpret_cast<char*>(&size), sizeof(size));
            // Append the actual string data.
            encodedString.append(str);
        }

        return encodedString;
    }

    // Decodes a single string to a list of strings by reading the fixed-size length prefix
    // and then reading the corresponding number of characters.
    vector<string> decode(const string& s) {
        vector<string> decodedStrings;

        size_t i = 0;
        int stringSize = 0;

        while (i < s.size()) {
            // Copy the size information at the current position.
            memcpy(&stringSize, s.data() + i, sizeof(stringSize));
            i += sizeof(stringSize);

            // Get the substr starting from the current position with the extracted length
            decodedStrings.push_back(s.substr(i, stringSize));
            i += stringSize;
        }

        return decodedStrings;
    }
};

// The Codec object usage example:
// Codec codec;
// vector<string> strs = codec.decode(codec.encode(strs));

```

Typescript Solution

```typescript
// Encodes a list of strings to a single string.
// Each string's length is stored as a fixed-size prefix before the actual string content.
function encode(strs: string[]): string {
    let encodedString = '';

    for (const str of strs) {
        // Convert the size to a 32-bit integer and add it to the result.
        const size = str.length;
        const buffer = new ArrayBuffer(4);
        const view = new DataView(buffer);
        view.setUint32(0, size);

        // Convert the ArrayBuffer to a string and append it to the result.
        encodedString += String.fromCharCode.apply(null, new Uint8Array(buffer));
        // Append the actual string data.
        encodedString += str;
    }

    return encodedString;
}

// Decodes a single string to a list of strings by reading the fixed-size length prefix
// and then reading the corresponding number of characters.
function decode(s: string): string[] {
    const decodedStrings: string[] = [];

    let i = 0;

    while (i < s.length) {
        // Create an ArrayBuffer and DataView representing the size of the next string.
        const buffer = new ArrayBuffer(4);
        const view = new DataView(buffer);

        // Convert the next 4 characters into the string size.
        for (let j = 0; j < 4; j++) {
            view.setUint8(j, s.charCodeAt(i + j));
        }

        const stringSize = view.getUint32(0);
        i += 4;

        // Get the substring starting from the current position with the extracted length
        const str = s.substring(i, i + stringSize);
        decodedStrings.push(str);
        i += stringSize;
    }

    return decodedStrings;
}

// Usage example:
// const encoded = encode(['hello', 'world']);
// const decoded = decode(encoded);

```

Time and Space Complexity

Encode function:

- **Time Complexity:** The encoding function iterates over all strings in the list, appending a 4-character length header to each string. The time complexity for appending operations in Python lists is `O(1)`. Assuming the average length of strings is `k`, and there are `n` strings, the total time complexity will be `O(nk)` since it needs to process each character in each string.

- **Space Complexity:** Space complexity for the encoding function would be `O(nk)` as well. This is due to the fact that it constructs a new string containing all of the individual strings with their 4-character headers. The output size is proportional to the total size of all input strings.

Decode function:

- **Time Complexity:** For the decode function, a single pass through the encoded string is made, extracting the length of each string and then the string itself. With the same assumption for average string length `k` and `n` strings, the time complexity will be `O(nk)` because for each string, the function reads 4 characters for size and then `k` characters for the actual string.

- **Space Complexity:** The decode function creates a list of strings resulting in the same `O(nk)` space complexity as for the encoding function, as it needs to store all the decoded strings in memory.