
Problem Statement:

Design an algorithm to encode a list of strings into a single string and decode it back to the original list. The encoded format must avoid conflicts with characters inside the strings themselves and be scalable for various input sizes, including empty strings.

Solution One

Use a delimiter (#) to separate metadata and the actual content of each string. Length prefixes are inserted to avoid ambiguity when reading the encoded string.

```python
def encode_strings(strs: List[str]) -> str:
    delimiter = '#'
    encoded = ''.join(f"{delimiter}{len(s)}{delimiter}{s}" for s in strs)
    return encoded

def decode_strings(encoded: str) -> List[str]:
    string_array = []
    i = 0

    while i < len(encoded):
            if (encoded[i] == '#'):
                i += 1

            digit = ''
            while (encoded[i].isdigit()):
                digit += encoded[i]
                i += 1

            i += 1
            string_len = int(digit)
            string_array.append(encoded[i : i + string_len])
            i += string_len

    return string_array
```

Step-by-Step Breakdown

1. **Input:** ["neet", "code", "love", "you"]
2. **Encoding:**
    - For each string, prefix the length, add a delimiter (#), and concatenate the string.

["neet", "code", "love", "you"]
↓ Encoding
#4#neet#4#code#4#love#3#you

3. **Decoding:**

- First identify the string length by reading the integer before the delimiter.
- Skip the next delimiter and extract the exact substring based on the identified length.
- After each extraction, the index (i) moves forward to continue decoding the next segment.

- encoded string: #4#neet#4#code#4#love#3#you
- i = 0 -> Skipping '#'
- i = 1 -> Reading digit "4" (length of first string)
- i = 3 -> Skipping '#'
- i = 4 -> Extracting "neet"
- i = 8 -> Skipping '#'
- i = 9 -> Reading digit "4" (length of second string)
- i = 11 -> Skipping '#'
- i = 12 -> Extracting "code"
- i = 16 -> Skipping '#'
- i = 17 -> Reading digit "4" (length of third string)
- i = 19 -> Skipping '#'
- i = 20 -> Extracting "love"
- i = 24 -> Skipping '#'
- i = 25 -> Reading digit "3" (length of fourth string)
- i = 27 -> Skipping '#'
- i = 28 -> Extracting "you"

4. **Output:** ["neet", "code", "love", "you"]
5. **Efficiency:**
    - **Time Complexity:** Encoding and decoding both operate in **O(n)** where n is the total length of all strings combined.
    - **Space Complexity: O(n)** extra space required for storing encoded/decoded data.

Solution 2

Instead of using a delimiter, use binary encoding for lengths followed by string content.

```python
def encode_binary(strs: List[str]) -> str:
    result = b''
    for s in strs:
        encoded_length = struct.pack('>I', len(s))
        result += encoded_length + s.encode('utf-8')
    return result

def decode_binary(encoded: str) -> List[str]:
    string_array = []
    i = 0

    while i < len(encoded):
        length = struct.unpack('>I', encoded[i:i + 4])[0]
        i += 4
        string_array.append(encoded[i:i + length].decode('utf-8'))
        i += length
    return string_array
```

Step-by-Step Breakdown

1. **Input:** ["neet", "code", "love", "you"]
2. **Encoding:** Convert each string's length to a 4-byte binary format.

   "neet" -> \x00\x00\x00\x04 "neet"
   "code" -> \x00\x00\x00\x04 "code"

3. **Decoding:** Concatenate the binary length headers with the strings.

   Encoded bytes:
   \x00\x00\x00\x04neet\x00\x00\x00\x04code\x00\x00\x00\x04love\x00\x00\x00\x03you

- i = 0 -> Read 4-byte length header `\x00\x00\x00\x04` (length 4)
- i = 4 -> Extracting "neet"
- i = 8 -> Read 4-byte length header `\x00\x00\x00\x04` (length 4)
- i = 12 -> Extracting "code"
- i = 16 -> Read 4-byte length header `\x00\x00\x00\x04` (length 4)
- i = 20 -> Extracting "love"
- i = 24 -> Read 4-byte length header `\x00\x00\x00\x03` (length 3)
- i = 28 -> Extracting "you"

4. **Output:** ["neet", "code", "love", "you"]
5. **Efficiency:**
   - **Time Complexity:** Encoding and decoding are both **O(n)**
   - **Space Complexity: O(n)** for encoded and decoded data.

Comparison

Aspect	Solution One	Solution Two
Efficiency	**O(n)**	**O(n)**
Memory Usage	Moderate	Slightly higher
Ease of Implementation	Simple	Moderate
Scalability	Good	Excellent
