User Settings in the Cloud

Interviewer Onboarding

- Original author: rice@
- Original link: rice: Encode a list of strings

This document is intended as a guide for interviewers using the interview question about how to encode a list of strings into a single string.

Instructions that should be verbally shared with the candidate are printed against a purple background.

General guidelines for how to conduct a pairing interview can be found at go/pairinginterviews, including a list of topics to cover in the intro.

Overview

The question is conveyed to the candidate using the script in the next section. For the purposes of the interviewer, the question can be boiled down to the following (using Java syntax):

Write a pair of functions:

```
String encode(List<String> inputs)

List<String> decode(String input)
```

such that:

`decode(encode(anyInputList))` always returns a list equal to **`anyInputList`**

This question tests the candidate's ability to perform basic string manipulation, and to think about issues around encoding and transmission of data in serial form. It may challenge their ability to navigate various levels of representation (e.g., a source-level escaped regular expression matching an escape sequence).

For the interviewer, it's important to note the following:

- The encode and decode functions don't share any state, i.e., you can't store the data in a shared object somewhere. To ensure this constraint is met, the script below describes the encode and decode functions as running in separate app instances (e.g., web app versus mobile app).

- If the candidate recognized this as serialization and is familiar with a standard serialization library (such as JSON, or Java or Python serialization), let them know their idea would work fine in practice, but for purposes of the interview you would like them to "roll their own" solution using only basic primitives.

- Be sure the candidate tests edge cases including:
    - Empty list (no strings at all)
        - `[]`
    - A List containing an empty string
        - `[""]`
    - Lists containing a mixture of empty and non-empty strings
        - `["", "abc", "def"], ["abc", "", "def"], ["abc", "", ""]`
    - Strings containing characters that are special to the encoding format
        - `["1,234"]` (for a format that prepends a numeric count)
        - `["abc\\\"def\\"]` (for a format that escapes quotes and backslashes)

For the Candidate

Read this script to the candidate:

Imagine you are writing an app that exists on multiple platforms (for example, web and mobile). Inside the app, there is a list of "user settings" that the user can type in. For our purposes, we'll say this is just an ordered list of strings. Maybe the first string is the user's name, the second string is their address, and the third string is their favorite color - we don't really care what they mean, we just want to keep them in the same order they are given to us.

Since the user is typing in the strings, we want to handle anything that is a legal string in [programming language chosen by the candidate] - they can contain any character, any sequence of characters, newlines, emojis, control characters, etc.

We want to store these user setting strings to our (imaginary) cloud service, so the user can have the same settings no matter which version of the app they log into. Unfortunately, our cloud service is very simple - it can only store one string for each of our users.

So, we decide we need a pair of functions, one function to convert the list of strings that the user types in into a single string that we can store to the cloud, and another function to convert that single string back into a list of strings.

*Your job is to come up with the format of the single string that we will store in our cloud service, and to write a pair of functions (we can call them **`encode`** and **`decode`**) to perform the conversions. We don't want to lose any data, so if we call **`encode`** on some strings, store the result in the cloud, then call **`decode`** on the cloud data, we will always end up with a list of*

strings that is identical to the list we started with.

For the Interviewer

Questions and Answers

- Is there a character I can use as a delimiter in my string that won't appear in the input?
    - No, we allow the user to type in any characters so there is no character that can't appear in the input (including newlines or control characters).

- What if I use a bunch of characters that are really unlikely to appear in the input?
    - While the characters you choose may be unlikely, this would introduce a vulnerability that an attacker could exploit to cause us to lose data. We want a solution that works in all situations.

- Can the input have no strings at all?
    - Sure, in fact that is a reasonable starting state within the app.

- Is an empty string allowed in the input?
    - Sure, the user might erase whatever they typed earlier.

- What if one of the strings in the input list is **`null`**?
    - We can ignore this case, or if you like you can throw an exception in the encoder when this happens.
    - (Or, if there is enough time) Try modifying your encoding format to encode **`null`** entries in the input so that the output will have **`null`** entries as well.

- Are the user settings a hash/dictionary?
    - Not exactly, there's no key for each setting. They are just a list of strings in some order that makes sense to the app.

- Do I need to worry about the size of the inputs, or about running out of memory?
    - You can assume that everything will fit in memory. But we don't want to gratuitously use a lot of unnecessary space.

- Is this like encryption?
    - No, you are defining an "encoding" format but it is not "encrypted" - it doesn't need to hide its contents.

- Is this like compression?
    - No, you don't need to try to reduce the number of characters you use in your encoding, although you are encouraged not to use more space than necessary.

Common Pitfalls and Hints

- Pitfall 1: Escaping woes
    - It can be challenging to write code that talks about escaping, because the string literals in the code themselves need to be escaped. Using functions like Java's **`replaceAll()`** that take regex inputs can compound the problem because regex special chars will need an additional level of escaping. One way out is to use a non-traditional escape character (say, **`%`**) that doesn't interact with source language or regex escaping.

- Pitfall 2: Excessive string concatenation
    - Solutions often build up strings using lots of concatenation. See if the candidate knows about tools such as Java's **`StringBuilder`** and if they understand the efficiency implications of using them versus simple concatenation (which can be $O(n^2)$).

- Pitfall 3: Thinking the question is impossible
    - Some candidates struggle with the idea that the input can contain any character, and they convince themselves that this means there is no way to use special characters in their encoded format, "hence" the question is impossible. You can encourage the candidate to think about real-life examples of encodings, for example how would they write an HTML page that talks about HTML syntax (they would need to use the escape sequence **`<`** to write about HTML tags; HTML frees up the & char by using the escape sequence **`&`** to replace **`&`** itself). Or if the candidate has ever worked with hardware or networking protocols, they might be able to use the idea of a packet header that is decoded before attempting to decode the body of the packet. If the candidate is really stuck, you could point out that a snippet of source code in the editor like **`["hello","world"]`** is simultaneously a representation of an array or list in the memory of the compiler's representation of the program, while at the same time it is just a simple sequence of characters in the memory of the editor.

- Pitfall 4: Searching for delimiters manually
    - Candidates may forget that they can call a standard function like **`String.indexOf()`** to find the next occurrence of a delimiter character, leading to overlay complex code with nested loops.

- Pitfall 5: Single-digit or fixed-length numbers
    - When encoding the lengths of input strings numerically, candidates may erroneously design their encoding to only support a single digit length value. Suggest a test case with a ten-character input string to see if they can notice their

mistake. Ensure that they place a non-digit delimiter after the length and that they understand why it's OK to use any non-digit delimiter regardless of the contents of the input strings.
    - Some candidates allocate a fixed length field (say, 5 digits) and pad as needed. While acceptable, you can encourage the candidate to be more space efficient.

Discussion for the Interviewer

A key to the first part of this question is to give the candidate some time to come up with an idea for a solution. Weaker candidates will try to convince themselves that it is impossible, or will get stuck on an idea that violates the constraints of the problem. Be firm about adhering to the constraints (no character is special, there is no shared memory between encoder and decoder).

Once they have an idea that can work, get them to write the encoder and test it. Then move on to the decoder which is usually more difficult. Now you can focus on end-to-end testing. Ask them to tell you whether a new test case will pass without running the code.

Occasionally a very strong candidate is able to complete the question extremely quickly. In this case you can encourage them to invent and implement an additional approach and compare them. However, the vast majority of candidates end up taking 40-50 minutes to come up with a clean, well-tested solution, even when they seem to start off quickly.

Typical Solutions

JSON Subset

Many candidates recognize that JSON is capable of performing the requested task. Fewer are familiar with precisely how JSON works. You can encourage the candidate to implement a JSON subset that only deals with an array of strings like: **`["hello", "world"]`**

A correct solution will use a backslash (or some other character) to escape double quotes and backslashes (or whatever escape character is chosen). The order of escaping matters, if the candidate tries this:

`input = input.replace("\"", "\\\"")` // replace quotes with backslash+quote
`input = input.replace("\\", "\\\\")` // replace backslashes with backslash+backslash

there will be too many backslashes since a quote will turn into 3 backslashes + 1 quote. The reverse order is correct.

A correct solution will be cautious about relying on a function like **`String.split()`**, which can be fooled by edge cases like:

`["hello\",","world"]` - cannot be split on the obvious string quote+comma+quote

A straightforward decoding solution is to scan the string from left to right, using boolean state variables to keep track of whether we are inside a quoted string and whether the previous character was a backslash. Attempting to scan backwards to determine whether a given character is escaped is possible but tricky, it's much easier to always scan forwards.

The main insights for this approach are that correct escaping requires two levels (you need to escape the escape character when it appears), that the order of escaping is significant, and that typical string **JOIN** and **SPLIT** primitives are not exact inverses of one another.

Simple Escaping

It's possible to use escaping without the rest of JSON syntax. Say we use a comma as a delimiter between strings and escape commas within the string (and escape backslashes as well). Then **`["hello","world"]`** becomes **`"hello,world"`** and **`["hi, user","good\\,bye"]`** becomes **`"hi\,user,good\\\,bye"`**.

A correct encoder will implement two levels of escaping (escaping the escape character).

The decoder requires parsing the input from left to right and setting a boolean **`escaped`** when we see a backslash. When **`escaped`** is true, we just append the next character to the current string and set escaped to false. When **`escaped`** is false and we see a comma, we push the current string onto the output list and set the current string to empty.

A variation is to escape the delimiter in such a way that it no longer contains the original delimiter, for example

```
input = input.replace("&", "&amp;")
input = input.replace(",", "&comma;")
```

Now the transformed input does not contain any commas. We can use **`String.split()`** to decode, then reverse the escaping for each piece:

```
input = input.replace("&comma;", ",")
input = input.replace("&amp;", "&")
```

You'll still want test cases to ensure that **split** does the right thing when the list is empty or when there are empty strings present (i.e, it does not coalesce multiple blank entries). The exact behavior will depend on what language/library is being used.

Counted Strings

These solutions encode the string lengths or starting positions numerically. The length/position data can appear at the start of the string or interspersed with the input strings. These solutions have the nice property that the input strings don't need to be modified and can simply be extracted as substrings from the encoded string:

`["hello", "world"]` → **`"5,hello5,world"`** or **`"5,5|helloworld"`** or **`"0,5|helloworld"`** (where 0 and 5 are the starting indexes of the two strings).

The encoding must separate the length/index from the data that follows using a non-digit separator character (i.e., **`"5,hello"`** not **`"5hello"`** which will fail on input strings of ten or more characters).

If the data takes the form of a header, the formation must also be able to signal the end of the header, either by an initial count (like **`"2,5,5,helloworld"`** which says there will be 2 strings of 5 characters each) or a distinct delimiter character (like **`"5,5|helloworld"`**).

If the candidate attempts to place the metadata at the end rather than at the beginning (i.e., a "footer"), enquire how they will be able to locate it. Typically this would require seeking backwards from the end to some special character that marks the start of the footer.

The main insight for this approach is that the encoder has sufficient control over its output format that it is able to make some characters special, even if they appear in the input strings.

Dynamic Delimiters

Some candidates want to use **JOIN** and **SPLIT** and are willing to do some work to find a suitable delimiter sequence by scanning the input. For example, they could generate a random string and reject it if it is a substring match for any input string, or they could have a loop to try a sequence like **`["AA", "AB", "AC", ..., "BA", "BB", ...]`**.

For such a solution to be correct:

- The encoder must provide a way for the decoder to know what delimiter it used, usually by prepending it to its output (followed by a known delimiter, e.g., **`"QXZ:helloQXZworld"`** uses a colon to indicate the end of the delimiter being used).

- The delimiter must be unambiguous, for example if we want to delimit `["ABC", "DEF"]` we can't use `CC` as a delimiter even though it doesn't appear in any of the input strings because the encoding `"ABCCCDEF"` is also the encoding of `["AB", "CDEF"]`.
- You should insist on the delimiter not exceeding a reasonable number of characters. You can have a discussion of how many characters are needed in order to guarantee the existence of an unambiguous delimiter for a set of input strings of given length. Something like a 36-character UUID would be extreme overkill.

Base64 Encoding

Another approach is to transform the input strings into a form that is guaranteed to leave some characters available to be used as delimiters. Base64 is a common form that ensures that its output only includes ASCII letters (upper and lower case), digits, and two other characters that may vary between implementations. Any character apart from that set is free to be used as a delimiter.

A pitfall with Base64 encoding is that some languages (notably Python) define it on byte arrays rather than strings, which requires the caller to be explicit about the character encoding used by the strings to be encoded. UTF8 will work. Avoid letting the candidate go too deeply into character encoding issues - they are really orthogonal to the question at hand which is about transforming from strings to strings, all in the source language's normal encoding.

For this approach, I usually try to see if the candidate has some understanding of how Base64 works and to get them to state explicitly why they want to use it (e.g., by saying something along the lines of "it's ok to use a comma between the strings and to split() on commas, because the base64 output can't contain any commas.") It's rare for a candidate to actually know enough about Base64 to be able to write their own codec. Some candidates are able to derive the fact that the encoding uses 4/3 of the original amount of storage (8 encoded bits are used to encode 6 raw bits).

This approach may yield less signal than some other approaches, because it is so straightforward once the mechanics of calling the Base64 library are in place. I try to make up for this by having more general discussion, and/or seeing if the candidate can at least outline some other possible approaches.

Expected Progress

- The first step is to design an encoding. This can involve a conversation about what characteristics we are looking for in terms of reliability, efficiency, and space used. Prompt the candidate with examples that break their scheme, and see if they can capture them into unit tests. The standard here is whether a human could decode the

format unambiguously. Don't allow the candidate to go further with a fundamentally unworkable scheme, but do allow subtle bugs that can be corrected later by suggesting test cases.

If the candidate suggests using JSON, protocol buffers, language-level serialization, etc., ask them if they can develop their own work-alike implementation rather than using the one provided by the system.

- The second step is to write the **`encode`** function and test it. These can be white-box tests that check against manually-generated output.
- The third step is to write the **`decode`** function and (white-box) test it.
- The fourth step is to write a black-box unit test for a round trip and call it with various inputs:

```
actual = decode(encode(expected))
assert actual.equals(expected)
```

- The fifth step is to discuss testing and to write a clean and comprehensive test suite, possibly using a standard testing framework such as JUnit (where CoderPad allows). Suggest tests that expose any remaining bugs and see if the candidate can locate and fix any underlying issues. Give the candidate an opportunity to run the code mentally against an edge case or two by asking a question like “Without actually running the code, do you think it will work on this input?”
- Additional optional steps could include hardening the decoder against invalid inputs, discussing time and space efficiency, and stubbing out an implementation of the simple cloud service.

How the Candidate Will Shine

- Clarifying questions
    - A strong candidate will ask questions and be able to repeat the key parts of the question back to show that they understand it fully.
- Complexity / space analysis
    - A strong candidate will demonstrate an understanding of how much space their encoding requires, and how much work is involved in building up strings in both the encoder and decoder.
- Algorithms
    - Depending on the approach, the solution may involve a small state machine or similar approaches from lexical analysis.
- Data structures
    - A weak candidate may try to convert the input into some other data structure other than a pure string. This demonstrates a lack of understanding of the

client/server setup of the question in which the encode and decode functions cannot share any data besides the one "cloud" string.

- Data types
    - A strong candidate will demonstrate their comfort with String/StringBuilder classes and character extraction functions. In some cases (e.g., Base64 solution in Python) they will demonstrate their knowledge of character set encoding.
- Error handling
    - When time allows, the **`decode`** function can be enhanced to detect impossible inputs.
- Tests
    - A strong candidate will produce a set of self-validating test cases that accumulate over the course of the interview. Weaker candidates will constantly modify an existing test case, and/or rely on visually inspecting the results rather than writing code to perform the comparison between expected and actual results.

Scaling with the problem

- A few candidates change approaches in midstream. Typically they think of the count-based approach after attempting an escaping-based approach. Allow these candidates to make the switch if there is enough time for them to be successful.

Elegant solutions

- To my taste, the most elegant solution is to alternate counts and data, e.g., **`"5,hello5,world"`**. This requires a minimal amount of code for both encode and decode, takes only a little storage space, is human-readable, and does not need to deal with escaping at all. Strings are extracted from the encoding using **`substring()`**. **`String.indexOf()`** and **`Integer.parseInt()`** do the work of finding the next delimiter and parsing out the length.

Special notes for languages/environments

- Python and Ruby coders can take advantage of higher-level constructs such as **`map`** and list comprehensions. Modern Java coders can take advantage of stream syntax.
- Don't allow JavaScript coders to rely on built-in object serialization.
- This question is especially hard in C. I've seen a solution involving a dynamic array of pointers to strings that essentially boiled down to manipulation of pointers to pointers to pointers to chars.
- For C/C++ coders, emphasize that the solution must use a string representation that allows **`nul`** (**`'\0'`**) characters, not the basic C string approach of **`nul`**-terminated strings since that violates the core constraint that we allow our inputs to contain all characters.

- Swift's support for string manipulation has evolved, and some candidates get into difficulty trying to perform operations whose syntax has been in flux such as extracting a character from a given position within a string.