393. UTF-8 Validation

Oct. 22, 2018 | 19.2K views

★★★★★ Average Rating: 4.83 (29 votes)

A character in UTF8 can be from **1 to 4 bytes** long, subjected to the following rules:

1. For 1-byte character, the first bit is a 0, followed by its unicode code.
2. For n-bytes character, the first n-bits are all one's, the n+1 bit is 0, followed by n-1 bytes with most significant 2 bits being 10.

This is how the UTF-8 encoding would work:

```
   Char. number range  |        UTF-8 octet sequence
      (hexadecimal)    |              (binary)
   --------------------+---------------------------------------------
   0000 0000-0000 007F | 0xxxxxxx
   0000 0080-0000 07FF | 110xxxxx 10xxxxxx
   0000 0800-0000 FFFF | 1110xxxx 10xxxxxx 10xxxxxx
   0001 0000-0010 FFFF | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
```

Given an array of integers representing the data, return whether it is a valid utf-8 encoding.

Note:
The input is an array of integers. Only the **least significant 8 bits** of each integer is used to store the data. This means each integer represents only 1 byte of data.

Example 1:

```
data = [197, 130, 1], which represents the octet sequence: 11000101 10000010 00000001.

Return true.
It is a valid utf-8 encoding for a 2-bytes character followed by a 1-byte character.
```

Example 2:

```
data = [235, 140, 4], which represented the octet sequence: 11101011 10001100 0000010

Return false.
The first 3 bits are all one's and the 4th bit is 0 means it is a 3-bytes character.
The next byte is a continuation byte which starts with 10 and that's correct.
But the second continuation byte does not start with 10, so it is invalid.
```

Solution

Intuition

This is an interesting problem to work with especially because it is not really hard to code up a solution for, but, you really need to pay attention to the details of the problem. A lot of people trying to solve the problem tend to miss out on small details that are mentioned and end up getting 1 or 2 test cases wrong.

Note: The following section provides 3 different examples for the problem and explains them. If the test cases and the rules are clear to you, you can skip over the `Approach 1`.

The problem statement provides 2 different examples for you to understand the rules to define a valid UTF-8 charset. That might not be enough for a lot of people and so the first thing we would do is try to understand all the rules given in the problem statement and in the meantime look at a few examples in detail that will help clarify the problem. Here are the rules in the question statement:

- A valid UTF-8 character can be `1 - 4` bytes long.
- For a `1-byte` character, the first bit is a `0`, followed by its unicode.
- For an `n-bytes` character, the first `n-bits` are all ones, the `n+1` bit is 0, followed by `n-1` bytes with most significant 2 bits being `10`.
- The input given would be an array of integers containing the data. We have to return if the data in the array represents a valid UTF-8 encoding. The important thing to note here is that the array doesn't contain data for **just a single character**. As can be seen from the first example, the array can contain data for multiple characters all of which can be valid UTF-8 characters and hence the charset represented by the array is valid.
- Another important thing to note is that the integers in the array can be larger than 255 as well. The highest number that can be represented by 8 bits is 255. So, what do we do if an integer in the array is say, 476? According to the Note in the problem before the examples, we only have to consider the **8 least significant bits** of each integer in the array.

Now that we have our rules defined for us, let us first look at the examples in the question and then some other examples from the discussion section that seem to cause a lot of confusion.

Example 1

```
data = [197, 130, 1]
```

Let us look at the octet sequence represented by the integers in this array. So, the octet sequence would be as follows:

```
11000101 10000010 00000001
```

> Remember, for an n-bytes UTF-8 character, the first n-bits would be 1 followed by a 0 in the n+1 bit. Then, the next n - 1 bytes would all have 10 as their most significant bits.

```
[1 1 0] 0 0 1 0 1
 ↑   ↑
```

Clearly, we can see that the 2 most significant bits of this byte are 1s and they are followed by a 0. This implies the start of a valid UTF-8 character. The information that we can gather from this byte is that this is a 2-byte UTF-8 character. This means that the next byte in the sequence must follow the pattern `10xxxxxx`. Let's see if it does.

```
[1 1 0] 0 0 1 0 1    [1 0] 0 0 0 0 1 0
 ↑   ↑                ↑ ↑
```

Yes, it does follow the intended sequence and hence the first two integers in the array i.e. `197 130` combine to form a valid 2-byte UTF-8 character. Since there are more elements left in the array, we move on and check them in a similar fashion as we did with the numbers above. The next integer in the array is `1`. Let's look at the binary representation for this integer.

```
00000001
```

Since the most significant bit itself of this number is a `0`, the only rule it satisfies is the 1-byte UTF-8 character rule. Let's re-iterate the rule:

> For 1-byte character, the first bit or the most significant but is a 0, followed by its unicode code.

```
[0] 0 0 0 0 0 0 1
 ↑
```

Clearly, the integer `1` is a valid 1-byte UTF-8 character in itself. Since there are no more elements left in the array to process, we will return `True` since there were two characters present in the array and both of them were valid UTF-8 encoded characters.

Example 2

```
[235, 140, 4]
```

This is the second example that's mentioned in the problem statement. As before, let us look at the binary representation of the integers in the array.

```
11101011 10001100 00000100
```

Let's start with the first integer in our array. The first byte will tell us the length of the UTF-8 character and hence the number of bytes we have to process in all in order to completely process a single UTF-8 character in the array before moving on to another one.

```
[1 1 1 0] 1 0 1 1
 ↑     ↑
```

So, the first few bits of the byte above are `1110`. This means that our UTF-8 character is of `3 bytes` in all. Remember the rule that helps us identify the size of a potential UTF-8 character from it's first byte.

> For an `n-bytes` character, the first `n-bits` are all one's, the `n+1` bit is 0.

Following this rule we determined that the first UTF-8 character is of 3 bytes. Since we are done processing one byte of data, we are left with 2 other bytes of data to process before starting with another UTF-8 character. Let's look at the remaining two bytes of the array.

```
[1 0] 0 0 1 1 0 0       0 0 0 0 0 1 0 0
 ↑ ↑                    ↑ (WRONG!)
```

The first byte above follows our pattern of `10xxxxxx` but the second byte does not. We had to verify a UTF-8 encoded 3-byte character as we saw from the first byte of the sequence `11101011`. The final byte is something that doesn't adhere to our rules mentioned before. Since we found an invalid byte, we can simply return `False` and we don't need to process any data further.

Example 3

We will look at one final example before moving onto the solution for this problem. This example has caused a lot of confusion as can be seen from multiple posts on the discussion forum:

- Discussion Post - 1
- Discussion Post - 2
- Discussion Post - 3

So, the example is:

```
[250,145,145,145,145]
```

Let us look at the binary representation of all the integers in the array.

```
11111010 10010001 10010001 10010001 10010001
```

As we have been doing in the previous two examples, let us look at the first byte of data to determine how many number of bytes our UTF-8 encoded character will have. Looking at the first byte of data we can see that our first UTF-8 encoded character in the sequence of data given, is of `5 bytes`.

```
[1 1 1 1 1 0] 1 0
 ↑         ↑
```

If this is a valid UTF-8 encoded character, the following four bytes of data should be in accordance with the pattern `10xxxxxx`. Let's look at the next 4 bytes of data one on each line.

```
1. [1 0] 0 1 0 0 0 1
2. [1 0] 0 1 0 0 0 1
3. [1 0] 0 1 0 0 0 1
4. [1 0] 0 1 0 0 0 1
```

As we can see above, all the 4 bytes are in accordance with the rules specified in the problem. Why then the result for this specific test case, `False`? People tend to miss out on one of the rules mentioned in the problem.

> This is the first rule in the problem statement and it clearly says that "A valid UTF-8 character can be 1 - 4 bytes long."

The first byte of data indicates that the UTF-8 encoded character contains `5 bytes` of data which cannot be true. This is why the answer for this specific test case is `False`.

Hopefully, most of your doubts would have been cleared by the three examples that we looked at above. Let us now move on to the solution(s) for this problem.

Approach 1: String Manipulation.

The problem itself is not that complicated. As long as we adhere to the rules specified in the problem, we should be fine. So, let's jump straight in and look at the algorithm.

Algorithm

1. Start processing the integers in the given array one by one.
2. For every integer, obtain the binary representation in the `string format`. Since integers can be very large, we should only keep/consider the `8 least significant bits` of data and discard the rest as mentioned in the problem statement. After this step, you should have 8-bits or 1-byte string representation for the integer. Let the string we get here be called `bin_rep`.
3. There are two scenarios that we need to consider here in the next step.
   1. One is that we are in the middle of processing some UTF-8 encoded character. In this case we simply need to check the first two bits of the string and see if they are `10` i.e. the 2 most significant bits of the integer being `1 and 0`. `bin_rep[:2] == "10"`
   2. The other case is that we already processed some valid UTF-8 characters and we have to start processing a new UTF-8 character. In that case we have to look at a prefix of the string representation and look at the number of `1`s that we encounter before encountering a `0`. This will tell us the size of the next UTF-8 character.
4. We keep on processing the integers of the array in this way until we either end up processing all of them or we find an invalid scenario.

Let us move on to the implementation of this algorithm.

Java Python Copy

```python
class Solution:
    def validUtf8(self, data):
        """
        :type data: List[int]
        :rtype: bool
        """

        # Number of bytes in the current UTF-8 character
        n_bytes = 0

        # For each integer in the data array.
        for num in data:

            # Get the binary representation. We only need the least significant 8 bits
            # for any given number.
            bin_rep = format(num, '#010b')[-8:]

            # If this is the case then we are to start processing a new UTF-8 character.
            if n_bytes == 0:

                # Get the number of 1s in the beginning of the string.
                for bit in bin_rep:
                    if bit == '0': break
                    n_bytes += 1

                # 1 byte characters
                if n_bytes == 0:
```

Complexity analysis

- Time Complexity : $O(N)$ since we process each integer of the array and for each integer we obtain an 8 character string which we then use for further processing. Overall the complexity is $O(N)$ considering N is the number of integers in the array.
- Space Complexity: $O(N)$ since for every integer we create a new string that we play around with.

Approach 2: Bit Manipulation

The previous solution is exactly what the problem asks us to do except that the string conversion and manipulation takes a lot of time and that is something unnecessary. We can make use of bit manipulation to perform the same task.

Let us look at what parts of a byte corresponding to an integer do we need to process.

1. If it is the starting byte for a UTF-8 character, then we need to process the first N bits where N will be at max 4. Anything more than that and we would have an invalid character.
2. In case the byte is a part of a UTF-8 character, then we simply need to check the first two bits or the most significant bits. The most significant bit needs to be a `1` and the second most significant bit needs to be a `0`.

Let's see how we can make use of bit manipulation to perform both of these tasks.

```
mask = 1 << 7
while mask & num:
    n_bytes += 1
    mask = mask >> 1
```

So, we have taken a mask = `1 << 7` which is basically `10000000`. We will make use of this mask and `logically and` it with the number to see if the bit at a particular position is set of not. We do this iteratively to check how many bits are set starting from the most significant bit (Remember, the integer might be too large but we should only process the 8 least significant bits of data.)

To check if the most significant bit is a `1` and the second most significant bit is a `0`, we can make use of the following two masks

```
mask1 = 1 << 7
mask2 = 1 << 6

if not (num & mask1 and not (num & mask2)):
    return False
```

The above code will simple use the `mask1` to check if the most significant bit is set to `1` and the second most significant bit is set to `0`. if this is not a case, then we return `False`.

Let's move onto the implementation.

Java Python Copy

```python
class Solution:
    def validUtf8(self, data):
        """
        :type data: List[int]
        :rtype: bool
        """

        # Number of bytes in the current UTF-8 character
        n_bytes = 0

        # Mask to check if the most significant bit (8th bit from the left) is set or not
        mask1 = 1 << 7

        # Mask to check if the second most significant bit is set or not
        mask2 = 1 << 6
        for num in data:

            # Get the number of set most significant bits in the byte if
            # this is the starting byte of an UTF-8 character.
            mask = 1 << 7
            if n_bytes == 0:
                while mask & num:
                    n_bytes += 1
                    mask = mask >> 1

                # 1 byte characters
                if n_bytes == 0:
```

Complexity analysis

- Time Complexity : $O(N)$.
- Space Complexity: $O(1)$.

Rate this article: ★★★★★

Previous Next

Comments: 5

Sort By

Type comment here... (Markdown is supported)

Preview Post

Miacova ★ 163 October 24, 2018 3:35 AM

For Algorithm 1, you say space complexity is O(N). I don't think that's correct. While you do allocate a string for each integer, you don't hold onto it. It's thrown away every loop. You're going to be using constant memory regardless of the string length. Having lots of allocations is bad but it's not what space complexity means.

13 Share Reply

SHOW 1 REPLY

RG0887 ★ 13 August 15, 2019 10:34 PM

Considering how confusing, bin manipulation problems can be, this is a very clear explanation of the problem statement and the solutions. Thank you!

3 Share Reply

JPV ★ 726 January 2, 2019 11:05 PM

Your problem specification, and the UTF-8 standard (https://tools.ietf.org/html/rfc3629) both define what set of codes can be encoded in each number of bytes. Your problem statement even has a big chart that says, "This is how the UTF-8 encoding would work:"

However, your "solution", and the online judge, fail to check this and fail to reject invalid encodings.

Read More

1 Share Reply

SHOW 1 REPLY

undefitied ★ 94 February 19, 2020 12:58 AM

Nice.

0 Share Reply

Laxmipooja ★ 5 January 23, 2020 4:46 AM

can anyone help why this is "True" and not "False"

[228,189,160,229,165,189,13,10]

0 Share Reply