# UTF-8 Validation Documentation

## Table of Contents

## 1. Problem Statement

We are given an integer array data where each integer represents a byte (0 to 255). Our goal is to determine whether the given sequence of bytes represents a valid UTF-8 encoding.

*A character in UTF-8 can be represented using 1 to 4 bytes, with the following rules:*

- ***1-byte character:*** The first bit is 0, followed by its Unicode code (0xxxxxxx).
- ***2-byte character:*** The first byte starts with 110, and the second byte starts with 10 (110xxxxx 10xxxxxx).
- ***3-byte character:*** The first byte starts with 1110, and the next two bytes start with 10 (1110xxxx 10xxxxxx 10xxxxxx).
- ***4-byte character:*** The first byte starts with 11110, and the next three bytes start with 10 (11110xxx 10xxxxxx 10xxxxxx 10xxxxxx).

We must ensure that the given data sequence follows these rules.

## 2. Intuition

Each UTF-8 character consists of a leading byte (determining length) and potential continuation bytes. By analyzing each byte's prefix, we can validate whether the sequence adheres to UTF-8 rules.

## 3. Key Observations

- The number of leading 1s in the first byte determines how many bytes the character should have.
- Continuation bytes should always start with 10 (10xxxxxx).
- If an invalid byte sequence is encountered, the input is not valid UTF-8.
- The algorithm should return True only if all bytes form valid UTF-8 sequences.

## 4. Approach

1. **Iterate through the data list and process each byte.**
2. **Determine the number of bytes in the current character:**
    - If the first byte starts with 0, it's a single-byte character.
    - Otherwise, count leading 1s to determine if it's a 2, 3, or 4-byte character.
3. **Validate the continuation bytes:**
    - Ensure the correct number of subsequent bytes start with 10 (10xxxxxx).
    - If any byte is invalid, return False.
4. **Return True if all characters are valid UTF-8 sequences.**

## 5. Edge Cases

- **Single-byte character:** [0], [127]
- **Multi-byte characters with valid structure:** [197, 130, 1] (valid 2-byte character followed by a 1-byte character)
- **Invalid continuation byte:** [235, 140, 4] (last byte is incorrect)
- **Leading byte not in range:** [255, 140, 140] (invalid first byte)
- **Not enough continuation bytes:** [240, 162] (should have 4 bytes but only 2 given)
- **Extra bytes without a leading byte:** [10, 10, 10] (invalid standalone continuation bytes)

## 6. Complexity Analysis

### Time Complexity

- O(N), where N is the number of bytes in data. We process each byte exactly once.

### Space Complexity

- O(1), as we use only a few integer variables for tracking remaining bytes.

## 7. Alternative Approaches

### Using Bit Manipulation for Faster Processing

Instead of using shifting operations, we can precompute masks and use bitwise AND to check patterns faster. However, the improvement is minimal compared to our current approach.

## 8. Test Cases

### Valid Cases

| Input | Expected Output |
|---|---|
| [0] | True |
| [197, 130, 1] | True |
| [240, 162, 138, 147] | True |

### Invalid Cases

| Input | Expected Output |
|---|---|
| [235, 140, 4] | False |
| [255, 140, 140] | False |
| [240, 162] | False |
| [10, 10, 10] | False |

## 9. Final Thoughts

This solution efficiently validates UTF-8 sequences using bitwise operations and a single pass over the input data. The approach ensures that all edge cases are handled correctly.

Further improvements could involve optimizing bitwise operations, but the current implementation is already optimal for competitive programming and real-world use.