157. Read N Characters Given Read4

Easy Array Interactive Simulation Leetcode Link

Problem Description

The problem is to simulate the behavior of reading characters from a file using a predefined API `read4`. The `read4` API reads up to 4 characters from a file and stores them into a buffer. It returns the number of actual characters read, which could be fewer than 4 if the end of the file is reached.

You need to implement a `read` function that reads `n` characters into a given buffer `buf`. The function should only use the `read4` method to interact with the file, meaning it cannot access the file content directly. The goal is to fill the buffer `buf` with `n` characters if they are available in the file, and return the count of characters actually read. If there are fewer than `n` characters left in the file when `read` is called, the function should read the remaining characters, write them to `buf`, and return the actual count.

Additionally, the function may be called multiple times with different values of `n`. The `read` function has to maintain the reading position in the file across multiple calls.

In summary, there are three key requirements to the problem:

1. Read characters from a file and store them in the provided buffer `buf`.
2. Use the given `read4` API function to read from the file.
3. Return the actual number of characters read, which should not exceed `n`.

Intuition

To solve this problem, we need to understand that the `read` function needs to interact with the `read4` function to access the file data. Since we can only read 4 characters at a time, we might not always read exactly `n` characters in one go. As a result, we'll generally need to call `read4` multiple times, accumulating characters until we've read as many as `n` or have reached the end of the file.

The solution should execute in a loop where each iteration attempts to read up to 4 new characters into a temporary buffer `buf4`, and then copy these characters to the final destination buffer `buf`. The loop needs to keep track of two main variables:

1. The number of characters that have been read and added to the destination buffer `buf`.
2. The number of characters read from the file in the current iteration using `read4`.

The loop must continue until we have either read `n` characters or there are no more characters to read from the file (i.e., `read4` returns fewer than 4 characters).

Each time `read4` is called, we iterate through the characters in `buf4` and add them to `buf`. If the destination buffer `buf` is filled with `n` characters, the function should return immediately, indicating that the requested number of characters has been read.

The variable `i` in the provided code snippet keeps track of the number of characters copied to `buf`, and it's incremented after each character is copied. When `i` equals `n`, or when `read4` does not return any new characters (indicating the end of the file), the function should return `i` as the total count of characters read.

Solution Approach

The solution provided follows an iterative approach, which effectively combines the usage of the buffer from `read4` and the main buffer `buf` where the final output will be stored. Here's a step-by-step explanation of how the solution works:

1. **Initialization**: Before entering the loop, an index `i` is initialized to keep track of the number of characters copied to the destination buffer `buf`. This `i` will be incremented after copying each character. A temporary buffer `buf4` with space for 4 characters is also created to hold the characters read from the file by `read4`.

2. **Reading from File**: Inside the `while` loop, `read4` is called, and its return value is stored in `v`. The return value represents the number of characters actually read from the file, which can be anything from 0 to 4.

3. **Copying to Destination Buffer**: After reading characters into `buf4`, a `for` loop iterates over the characters in `buf4`. For each character, the following is done:

   - Copy from `buf4` to `buf` at the current index `i`.
   - Increment `i`.
   - Check if `i` is equal to or greater than `n`, which means we've read the required number of characters. If true, we return `n` immediately, as we've fulfilled the request.
4. **Ending the Loop**: The `while` loop checks if `v` is at least 4, indicating that there might be more characters to read. If `v` is less than 4, it means the end of the file has been reached, or there are not enough characters left to fill `buf4` completely, and the loop ends.

5. **Returning Read Characters**: Outside the loop, after reading all necessary characters or reaching the end of the file, `i` (which represents the actual count of characters copied to `buf`) is returned.

The algorithm used here is straightforward and leverages both the `read4` API constraint (i.e., reading up to only 4 characters at a time) and the requirement to copy a precise number of characters to the destination buffer `buf`. No complex data structures are required, and the pattern is to iterate and copy until the condition is satisfied.

This solution is robust and easy to understand. It calculates the right amount of characters needed and uses minimal extra space, only requiring an additional small buffer `buf4` to bridge between `read4` and the final `buf`.

Example Walkthrough

Let's go through an example to see how the solution approach works. Imagine we have a file with the content "LeetCode" and we want to read 6 characters from it. We will use the `read` method to accomplish this task.

Consider the case where the `read` function is used as follows: `read(buf, 6)`. We start with `buf` being an empty reference to a character array.

1. **Initialization**: A temporary buffer `buf4` is created to store the characters read from the file by `read4`, and an index `i` is initialized to 0.

2. **First Call to `read4`**:

   - `read4` is called, returning a value of 4, since it reads "Leet" from the file.
   - Now, the `buf4` contains "Leet".
3. **Copying to Destination Buffer (First Loop)**:

   - Characters from `buf4` are copied to `buf` one by one.
   - `i` is incremented with each character copied. After copying "Leet", `i` is now 4.
4. **Check if More Characters are Needed**: Since `i` (4) is less than `n` (6), the loop continues.

5. **Second Call to `read4`**:

   - `read4` is called again, now it reads "Code" from the file.
   - The `buf4` contains "Code", and `read4` returns 4, but we only need 2 more characters.
6. **Copying to Destination Buffer (Second Loop)**:

   - We only copy two characters because `i` needs to reach 6.
   - After copying "Co", `i` is now 6, which is equal to `n`.
7. **End Loop and Return**: Since `i` has reached `n`, we stop and return `i`.

Therefore, after the `read(buf, 6)` call, `buf` contains "LeetCo", and the function returns 6 because that's the number of characters we wanted to read.

Remember, the actual buffer `buf` is intended to be large enough for the `n` characters, and the values are not shown as a string but copied into it like an array. The provided explanation simplifies the representation for clarity.

Python Solution

```python
class Solution:
    def read(self, buf, n):
        """
        Reads up to n characters from a file using read4() and stores them into buf.

        :param buf: Destination buffer (List[str]) where characters are to be stored
        :type buf: List[str]
        :param n: Number of characters to read
        :type n: int
        :return: The actual number of characters read
        :rtype: int
        """

        # Initialize the index pointer for the output buffer
        index = 0

        # Temporary buffer to store the output of read4
        buf4 = [''] * 4

        # Variable to store the number of characters read in last read4 operation
        chars_read = 4  # Initialize to 4 to enter the while loop

        # Continue reading until we have read 4 or fewer chars (end of file)
        while chars_read == 4:
            # Read the next 4 (or fewer) chars from the file into buf4
            chars_read = read4(buf4)

            # Iterate over the chars read into buf4
            for i in range(chars_read):
                # Copy the character from buf4 to the destination buffer
                buf[index] = buf4[i]
                index += 1  # Increment the index in the destination buffer

                # If we have read the required number of characters, return n
                if index == n:
                    return index

        # Return the actual number of characters read
        return index

```

Java Solution

```java
public class Solution extends Reader4 {
    /**
     * Reads up to 'n' characters from the file and stores them in 'buf'.
     *
     * @param buf Destination buffer to store read characters.
     * @param n Number of characters to read.
     * @return The number of actual characters read, which might be less than 'n' if the file has fewer characters.
     */
    public int read(char[] buf, int n) {
        // Temporary buffer to hold chunks of read characters
        char[] tempBuffer = new char[4];

        // Index for the destination buffer 'buf'
        int bufIndex = 0;

        // Variable to hold the count of characters actually read in each read4 call
        int charReadCount = 0;

        // Continue reading until there are fewer than 4 characters returned, which signifies end of file or buffer
        do {
            // Read up to 4 characters into tempBuffer
            charReadCount = read4(tempBuffer);

            // Copy characters from tempBuffer to buf, up to the number of characters requested 'n'
            for (int j = 0; j < charReadCount; ++j) {
                buf[bufIndex] = tempBuffer[j];
                bufIndex++;

                // If 'bufIndex' reaches 'n', we've read the required number of characters
                if (bufIndex == n) {
                    return n; // The requested number of characters have been read
                }
            }
        } while (charReadCount == 4); // Continue if we read 4 characters, meaning there could be more to read

        // Return the number of characters actually stored in 'buf'
        return bufIndex;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    /**
     * Reads characters into buf from a file and returns the actual number
     * of characters read, which could be less than n if the end of file is reached.
     * @param buf - Destination buffer to store read characters
     * @param n   - Number of characters to be read
     * @return    - Actual number of characters read
     */
    int read(char* buf, int n) {
        char tempBuffer[4]; // Temporary buffer to hold read chunks of 4 characters
        int totalCharsRead = 0; // Total characters read

        while (true) {
            // Read up to 4 characters into tempBuffer from file
            int charsRead = read4(tempBuffer);

            // Transfer characters from tempBuffer to destination buf
            for (int j = 0; j < charsRead; ++j) {
                buf[totalCharsRead++] = tempBuffer[j];

                // If the number of characters requested (n) is reached,
                // return the number of characters read so far.
                if (totalCharsRead == n) {
                    return n;
                }
            }

            // Break the loop if we read less than 4 characters,
            // which means end of file is reached
            if (charsRead < 4) {
                break;
            }
        }

        // Return total number of characters actually read
        return totalCharsRead;
    }
};

```

Typescript Solution

```typescript
// Assuming read4 is already defined elsewhere to read 4 characters at a time from the file
declare function read4(tempBuffer: string[]): number;

/**
 * Reads characters into buf from a file and returns the actual number of characters read,
 * which could be less than n if the end of file is reached.
 * @param buf - Destination buffer to store read characters
 * @param n - Number of characters to be read
 * @return - Actual number of characters read
 */
let read = (buf: string[], n: number): number => {
    // Temporary buffer to hold read chunks of 4 characters
    let tempBuffer: string[] = [];
    // Total characters read
    let totalCharsRead = 0;

    while (true) {
        // Read up to 4 characters into tempBuffer from file
        let charsRead = read4(tempBuffer);

        // Transfer characters from tempBuffer to destination buf
        for (let j = 0; j < charsRead; ++j) {
            // Store the character in the destination buffer
            buf[totalCharsRead++] = tempBuffer[j];

            // If the number of characters requested (n) is reached,
            // return the total number of characters read so far.
            if (totalCharsRead === n) {
                return n;
            }
        }

        // Break the loop if we read less than 4 characters, signaling end of file
        if (charsRead < 4) {
            break;
        }
    }

    // Return total number of characters actually read
    return totalCharsRead;
};

```

Time and Space Complexity

Time Complexity

The time complexity of the code is determined by the number of times `read4` is called and the number of times the inner loop runs. The `read4` function is called until it returns less than 4 characters, which indicates the end of the file or the buffer is fully read.

The maximum number of times `read4` can be called is `ceil(n/4)`, since `read4` reads 4 characters at a time and `n` is the total number of characters we want to read. In the worst case, the inner loop runs 4 times for each call to `read4` (if `read4` returns 4 characters each time). Therefore, the inner loop iteration count is at most 4 multiplied by the number of `read4` calls, giving us a potential maximum of `4 * ceil(n/4)` iterations.

However, due to the condition `if i >= n` inside the inner loop, the actual read process will stop as soon as `n` characters are read. Thus, the tight bound on the number of iterations of the inner loop is `n`. Therefore, the time complexity of the code is `O(n)`.

Space Complexity

The space complexity of the algorithm is determined by the additional space used by the algorithm besides the input and output. The additional space in this algorithm is utilized for the `buf4` array, which is a constant size of 4 characters.

No other additional space is growing with the input size `n`. Hence the space complexity is constant, or `O(1)`.