299. Bulls and Cows

Medium Hash Table String Counting

Leetcode Link

Problem Description

The "Bulls and Cows" game is a brain teaser where one player, the codemaker, chooses a secret number, and another player, the codebreaker, tries to guess the number. The guess is made up of digits, and for each guess, the codemaker must provide a hint in two parts:

- The number of "bulls", which are digits that are correctly placed in the guess.
- The number of "cows", which are digits that are present in the secret number but are not in the correct position. Our goal is to create a function that takes the secret number and the guess as inputs and returns the hint according to the rules of the game. The hint should be a string formatted as `"xAyB"`, where `x` represents the number of bulls and `y` represents the number of cows.

Intuition

To solve the "Bulls and Cows" problem, we need to compare the `secret` and the `guess` strings character by character to count the bulls. A bull is when both have the same digit at the same index. When we encounter a bull, we increment our `bulls` count by one. For cows, we need to consider the digits that are present in both `secret` and `guess` but are not bulls. To manage this, we keep two separate counts (arrays) of the digits (0 through 9) that appear in `secret` and `guess` but are not bulls.

Once the bulls' count is complete, we can iterate over the counts array to calculate the cows by finding the minimum count of each digit between the `secret` and the `guess`. The idea is that a cow can only happen if a digit is present in both `secret` and `guess`; the number of possible cows for a particular digit is the smaller count of that digit in both strings (since a digit can't be moved around more times than it appears).

The overall intuition behind the solution is to:

1. Identify and count the bulls as we iterate through the strings.
2. Count the non-bull digits for both strings.
3. The cows count is the sum of the minimum counts of each digit between `secret` and `guess`.

By the end of the iteration, we will have the number of bulls and cows, which we can then format as a string and return as per the game's rules.

Solution Approach

The implementation of the solution in Python involves iterating over the `secret` and `guess` strings and using two auxiliary arrays to keep track of non-bulls digits.

The Solution class has a method `getHint` with two parameters: `secret` and `guess`. Let's walk through the steps of the method:

1. We initialize `x` and `y` as integers to zero, where `x` will count the number of bulls and `y` will count the number of cows.
2. Two lists, `cnt1` and `cnt2`, each of size 10 (to account for digits 0-9), are initialized to count the occurrences of each digit in `secret` and `guess`, respectively, excluding those that are already bulls.
3. A `for` loop goes through each index `i` of the `secret` and `guess` strings. If the characters at the current index match, we increment the `x` counter by one, indicating a bull has been found.
4. If the digits do not match (no bull is found), we increment the counts in `cnt1` and `cnt2` corresponding to the digit in `secret` and `guess` at index `i`. This accounts for potential cows.
5. After we have completed the iteration for bulls, we loop through the digits from 0 to 9 to calculate the number of cows. For each digit, we take the minimum count found in `cnt1` and `cnt2`. This minimum count represents how many times a non-bull digit in `guess` could potentially be a bull if rearranged, hence a cow. We sum up these minimums to get the total number of cows (`y`).
6. Finally, we format the output as a string that shows the number of bulls (`x`) followed by 'A' and the number of cows (`y`) followed by 'B'. This is done using a formatted string (an f-string) `f'{x}A{y}B'` which is then returned by the method.

The algorithm is efficient because it requires only a single pass to count bulls, and a second pass through a fixed-size list of digits to count cows, making the time complexity linear in terms of the length of the secret string. The use of two counting arrays also efficiently handles situations where digits appear multiple times in the secret and guess.

Here's part of the code that handles the counting of bulls and updating the counts:

```
for i in range(len(secret)):
    if secret[i] == guess[i]:
        x += 1
    else:
        cnt1[int(secret[i])] += 1
        cnt2[int(guess[i])] += 1
```

And here's the code for calculating cows from the counts:

```
for i in range(10):
    y += min(cnt1[i], cnt2[i])
```

The final result is formatted and returned as such:

```
return f'{x}A{y}B'
```

This approach is a simplistic yet effective way of solving the "Bulls and Cows" game problem systematically and efficiently using basic concepts of iteration and frequency counting.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Assume the `secret` number chosen by the codemaker is `"1807"` and the `codebreaker` makes a guess `"7810"`. We'll go through each step of the method to determine the number of bulls and cows.

1. Initialize variables `x` and `y` to zero, where `x` will count bulls, and `y` will count cows.
2. Initialize two lists, `cnt1` and `cnt2`, each of size 10 and filled with zeros. These lists will count the occurrences of digits from 0 to 9 in `secret` and `guess`, excluding bulls.
3. Iterate through each index `i` of `secret` and `guess`:
   - At index 0, `secret` has `1` and `guess` has `7`, these are not bulls, so we increase `cnt1[1]` by 1 and `cnt2[7]` by 1.
   - At index 1, `secret` has `8` and `guess` has `8`, this is a bull, so we increase `x` by 1.
   - At index 2, `secret` has `0` and `guess` has `1`, these are not bulls, so we increase `cnt1[0]` by 1 and `cnt2[1]` by 1.
   - At index 3, `secret` has `7` and `guess` has `0`, these are not bulls, so we increase `cnt1[7]` by 1 and `cnt2[0]` by 1.
4. After looping through the strings, we now calculate the cows by iterating through the range of digits 0 to 9:
   - For `0`, `cnt1[0]` is 1 and `cnt2[0]` is 1, so `y` increases by `min(1, 1)` which is 1.
   - For `1`, `cnt1[1]` is 1 and `cnt2[1]` is 1, so `y` increases by `min(1, 1)` which is 1.
   - For `7`, `cnt1[7]` is 1 and `cnt2[7]` is 0, so `y` does not increase because `min(1, 0)` is `0`.
   - Other digits `cnt1` and `cnt2` are `0`, they don't contribute to `y`.
5. By summing all the minimums, we calculate the `y` as 2 (for the digits `8` and `1`).
6. Finally, the `x` count is 1 (for the digit `8`) and the `y` count is 2, so the formatted string output becomes `"1A2B"`.

This example demonstrates the effectiveness of the algorithm. Despite having the right digits in the wrong places (`7` and `0`), it effectively computes the correct counts. This follows directly from the insight that a cow occurs only if a digit is present in both `secret` and `guess`, regardless of its place, excluding the bulls that were identified earlier in the process. The game hints for the codebreaker would then be "1 bull, 2 cows."

Python Solution

```python
class Solution:
    def getHint(self, secret: str, guess: str) -> str:
        # Initialize the counters for bulls ('A') and cows ('B')
        bulls = cows = 0

        # Initialize the counters for the numbers that are not bulls
        secret_count = [0] * 10
        guess_count = [0] * 10

        # Iterate over both the secret and the guess strings
        for index in range(len(secret)):
            # Check if the current digits in both strings are the same (bulls)
            if secret[index] == guess[index]:
                bulls += 1  # Increment bulls counter
            else:
                # If it not a bull, increment the counts for the unmatched digits
                secret_count[int(secret[index])] += 1
                guess_count[int(guess[index])] += 1

        # Iterate over the digits from 0 to 9
        for i in range(10):
            # Number of cows is the minimum of the same digit's count in both the secret and the guess
            cows += min(secret_count[i], guess_count[i])

        # Return the formatted hint string
        return f'{bulls}A{cows}B'
```

Java Solution

```java
class Solution {
    public String getHint(String secret, String guess) {
        // Initial count of bulls ("A") and cows ("B")
        int bulls = 0, cows = 0;
        // Arrays to keep count of numbers not matched (bulls) in secret and guess
        int[] secretCount = new int[10];
        int[] guessCount = new int[10];

        // Iterate through each character in secret and guess strings
        for (int i = 0; i < secret.length(); ++i) {
            int secretDigit = secret.charAt(i) - '0'; // Convert char to int
            int guessDigit = guess.charAt(i) - '0'; // Convert char to int

            // If digits match at the same index, it's a bull ("A")
            if (secretDigit == guessDigit) {
                ++bulls;
            } else {
                // If not a bull, tally in corresponding count arrays
                ++secretCount[secretDigit];
                ++guessCount[guessDigit];
            }
        }

        // Calculate cows ("B") by finding the minimum count of each digit in both secret and guess that were not bulls
        for (int i = 0; i < 10; ++i) {
            cows += Math.min(secretCount[i], guessCount[i]);
        }

        // Return the formatted string with bulls ("A") and cows ("B")
        return String.format("%dA%dB", bulls, cows);
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    string getHint(string secret, string guess) {
        // Initialize bull and cow counters
        int bulls = 0, cows = 0;

        // Arrays to count the occurrence of each digit in secret and guess
        vector<int> countSecret(10, 0);
        vector<int> countGuess(10, 0);

        // Loop through the strings to find bulls and to count numbers
        for (int i = 0; i < secret.length(); ++i) {
            int secretDigit = secret[i] - '0', guessDigit = guess[i] - '0';

            // If digits match, increment bulls
            if (secretDigit == guessDigit) {
                ++bulls;
            } else {
                // Otherwise, increment the count of the digit for each string
                ++countSecret[secretDigit];
                ++countGuess[guessDigit];
            }
        }

        // For digits that don't match, calculate cows
        for (int i = 0; i < 10; ++i) {
            // The number of cows is the minimum of the same digit in both strings
            cows += min(countSecret[i], countGuess[i]);
        }

        // Return the result in the format "xBulls and yCows"
        return to_string(bulls) + "A" + to_string(cows) + "B";
    }
};
```

Typescript Solution

```typescript
// Function to calculate the hint based on the secret and guess strings
function getHint(secret: string, guess: string): string {
    // Initialize bull and cow counters
    let bulls: number = 0;
    let cows: number = 0;

    // Arrays to count the occurrence of each digit in secret and guess
    let countSecret: number[] = new Array(10).fill(0);
    let countGuess: number[] = new Array(10).fill(0);

    // Loop through the strings to find bulls and to count digits
    for (let i = 0; i < secret.length; i++) {
        let secretDigit = parseInt(secret[i], 10);
        let guessDigit = parseInt(guess[i], 10);

        // If digits match, increment bulls
        if (secretDigit === guessDigit) {
            bulls++;
        } else {
            // Otherwise, increment the count of the digit for each string
            countSecret[secretDigit]++;
            countGuess[guessDigit]++;
        }
    }

    // For digits that don't match, calculate cows
    for (let i = 0; i < 10; i++) {
        // The number of cows is the minimum of the same digit in both strings
        cows += Math.min(countSecret[i], countGuess[i]);
    }

    // Return the result in the format "xBulls and yCows"
    return `${bulls}A${cows}B`;
}
```

Time and Space Complexity

Time Complexity

The provided code involves iterating over the length of the `secret` and `guess` strings exactly once. The `for` loop runs for `len(secret)` times, and the operations inside the loop have a constant time complexity. After that, there is another loop that iterates a fixed number of times (10), as the possible digits are 0-9 for any guessing game. This loop also performs operations with a constant time complexity.

Therefore, the time complexity is determined by the length of the `secret` string. Let's denote the length of the `secret` string as n. The time complexity of the code is $O(n) + O(10)$, which simplifies to $O(n)$.

Space Complexity

Two arrays, `cnt1` and `cnt2`, each with a size of 10, are used to store the counts of digits from `0` to `9` in `secret` and `guess`, respectively. The size of these arrays is constant and does not depend on the input size. No additional space that scales with the size of the input is being used. Thus, the space complexity of the code is $O(1)$ for the constant space used by the counts of the digits.