771. Jewels and Stones

Easy Hash Table String Leetcode Link

Problem Description

In this problem, we are given two strings, `jewels` and `stones`. The `jewels` string represents the types of stones that are considered to be jewels. Each character in the `jewels` string is a unique jewel type. The `stones` string contains the stones that we have, and each character represents a type of stone. Our task is to count how many stones we have that are also jewels. It is important to note that the identification of jewels is case sensitive, meaning `a` and `A` would be treated as different types of stones.

Intuition

The solution approach involves using a set data structure in Python, which contains only unique items and allows for fast membership testing. By converting the `jewels` string into a set, we can quickly check if a stone (character in `stones` string) is a jewel by seeing if it exists in the set.

Here is the intuition behind the steps:

1. Convert the `jewels` string into a set. This is important to remove any duplicate characters, which can occur in the input, and to allow for O(1) (constant time) look-ups.
2. Iterate through each character in the `stones` string. For every stone, we will check if it's also a jewel.
3. We check if a stone is a jewel by determining if that stone (character) is present in the `jewels` set.
4. Count the number of stones that are jewels. This is done by summing the number of True values yielded by the expression `c in s` for each stone `c` in the `stones` string, where `s` is the set of jewels.
5. Return the count as the final answer.

The reasoning behind this approach is centered on set theory—by comparing elements of one set (our stones) to another (our jewels), we're looking for the intersection—the items that are common to both. The use of a set data structure simplifies this search, making the algorithm efficient and concise.

Solution Approach

The implementation of the solution relies on several concepts and data structures, which optimize the process of identifying jewels within the stones. Let's dive into the approach based on the reference solution:

1. **Set Data Structure**: The solution begins with creating a set `s` from the `jewels` string. Sets are an appropriate choice for two main reasons: they inherently prevent duplicate elements and provide constant time complexity (`O(1)`) for membership checking. This is crucial because it prevents the algorithm's performance from degrading, even with a large number of jewel types.

   ```python
   s = set(jewels)
   ```

2. **For Loop and Membership Testing**: We iterate through each character `c` in the string `stones`, which allows us to examine each stone we possess.

   ```python
   for c in stones:
   ```

3. **Boolean Expression in Sum**: For every stone represented by character `c`, we test if it is in the set `s` (the jewels). The expression `c in s` evaluates to a boolean (`True` if `c` is a jewel, `False` if not).

   ```python
   c in s
   ```

4. **Sum to Count Trues**: The sum function in Python conveniently treats `True` as `1` and `False` as `0`. Thus, when we sum up the boolean expression results, we effectively count how many times `c in s` evaluates to `True`, which corresponds to the number of stones that are jewels.

   ```python
   sum(c in s for c in stones)
   ```

5. **Return the Result**: The result of the sum operation is the final answer, which is the number of stones that are also jewels.

Combining these steps in a single expression leads to a compact and efficient solution. The solution's elegance lies in its use of Python's set and iteration mechanisms to directly tie together the question (is this a jewel?) with the answer (how many jewels?).

```python
class Solution:
    def numJewelsInStones(self, jewels: str, stones: str) -> int:
        s = set(jewels)
        return sum(c in s for c in stones)
```

This Python code snippet achieves our goal using advanced concepts in an easy-to-understand and concise way.

Example Walkthrough

Let's take a simple example to illustrate the solution approach. Suppose we have the following inputs:

- `jewels`: `"aA"`
- `stones`: `"aAAbbbb"`

As per the problem statement, each character in the `jewels` string is a unique jewel type, and they are case-sensitive. So in this case, we have two types of jewels: `'a'` and `'A'`. We have to find out how many of the stones are actual jewels.

Following the steps from the solution approach:

1. Convert the `jewels` string into a set. Thus, we create a set of jewels `s`:

   ```python
   s = set('aA')  # The set will contain {'a', 'A'}
   ```

2. Iterate through each character in the `stones` string using a for loop:

   ```python
   for c in "aAAbbbb":
   ```

3. For each character `c` in `stones`, check if it is present in the set of jewels `s`. We'll do this using the expression `c in s`, which returns `True` if `c` is a jewel, otherwise `False`:

   ```python
   'a' in s  # True
   'A' in s  # True
   'A' in s  # True
   'b' in s  # False
   'b' in s  # False
   'b' in s  # False
   'b' in s  # False
   ```

4. We evaluate the expression `c in s` inside a `sum()` function to add up the `True` values (which are counted as `1`):

   ```python
   sum(c in s for c in "aAAbbbb")  # This will equal 3
   ```

5. The result of the sum expression (3 in this case) is the number of stones that are also jewels, which is what we return as the answer.

Following this process, we were able to determine that out of the stones we possess (`"aAAbbbb"`), three of them are jewels (`"aAA"`).

Python Solution

```python
class Solution:
    def numJewelsInStones(self, jewels: str, stones: str) -> int:
        # Convert the string jewels into a set for faster lookup
        jewel_set = set(jewels)

        # Count how many characters in stones are also in jewel_set
        # The expression (char in jewel_set) returns True if the char is a jewel,
        # and False otherwise. sum() treats True as 1 and False as 0.
        jewel_count = sum(char in jewel_set for char in stones)

        # Return the total count of jewels in the stones
        return jewel_count

```

Java Solution

```java
class Solution {
    public int numJewelsInStones(String jewels, String stones) {
        // An array to keep track of all the ASCII characters.
        // The size is 128 as there are 128 ASCII characters.
        int[] asciiMap = new int[128];

        // Iterate over the characters in 'jewels' and mark them in the asciiMap.
        for (char jewel : jewels.toCharArray()) {
            asciiMap[jewel] = 1;
        }

        // 'totalJewels' holds the count of jewels found in 'stones'.
        int totalJewels = 0;

        // Iterate over the characters in 'stones'.
        for (char stone : stones.toCharArray()) {
            // If a character in 'stones' is marked as 1 in asciiMap,
            // it is a jewel, so we increment the totalJewels count.
            totalJewels += asciiMap[stone];
        }

        // Return the total count of jewels found in 'stones'.
        return totalJewels;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int numJewelsInStones(string jewels, string stones) {
        // Array to store whether a character is considered a jewel.
        // The ASCII value of the character will serve as the index.
        int jewelFlags[128] = {0};

        // Mark the jewels in the 'jewelFlags' array.
        for (char jewelChar : jewels) {
            jewelFlags[jewelChar] = 1;
        }

        // Count variable to keep track of the number of jewels found in 'stones'.
        int jewelCount = 0;

        // Iterate through each character in 'stones' to see if it's marked as a jewel.
        for (char stoneChar : stones) {
            // Increase count if current character is marked as a jewel.
            jewelCount += jewelFlags[stoneChar];
        }

        // Return the total count of jewels present in 'stones'.
        return jewelCount;
    }
};

```

Typescript Solution

```typescript
/**
 * Counts how many stones you have that are also jewels.
 * @param {string} jewels - The string representing the types of jewels.
 * @param {string} stones - The string representing the stones you have.
 * @returns {number} The number of stones that are also jewels.
 */
function numJewelsInStones(jewels: string, stones: string): number {
    // Create a Set to store unique jewel characters for quick lookup
    const jewelSet = new Set<string>([...jewels]);

    // Initialize a count variable to keep track of jewels found in stones
    let count = 0;

    // Iterate over the stones string to check if a stone is also a jewel
    for (const stone of stones) {
        // If the current stone is in the jewel set, increment the count
        if (jewelSet.has(stone)) {
            count++;
        }
    }

    // Return the total count of jewels found in stones
    return count;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n + m)` where `n` is the length of the `jewels` string and `m` is the length of the `stones` string. The creation of the set `s` from the `jewels` string takes `O(n)` time. Then, we iterate over each character `c` in `stones`, which takes `O(m)` time, to check whether it is in the set `s`. Since set lookup is `O(1)` on average, the time for the sum operation is also `O(m)`.

Space Complexity

The space complexity is `O(n)`, where `n` is the length of the `jewels` string. This is because we create a set `s` to store the distinct characters in `jewels`. There is no additional significant space used as we only store a single set.