205. Isomorphic Strings

Easy | Hash Table | String

Leetcode Link

Problem Description

In this problem, we are given two strings `s` and `t`. Our task is to determine if the two strings are isomorphic. Isomorphic strings are strings where each character in one string can be consistently replaced with another character to result in the second string. It is crucial to note that in isomorphic strings, the replacement of characters is done in such a way that the sequence of characters is maintained. Also, no two distinct characters can map to the same character, but a character is allowed to remap to itself.

In simpler terms, the pattern of characters in both strings should align. If 'x' in string `s` is replaced by 'y' in string `t`, then all 'x's in `s` must correspond to 'y's in `t`, and no other character in `s` should map to 'y'.

Intuition

To determine if `s` and `t` are isomorphic, we establish a mapping for each character in `s` to its corresponding character in `t`. This can be done using two hash tables or arrays, here indexed by the character's ASCII code, ensuring no collision for character storage.

Each array `d1` and `d2` records the latest position (index + 1 to account for the zeroth position) where each character is encountered in `s` and `t`, respectively. We iterate the strings simultaneously and compare the mapping indices.

If for any character pair `a` from `s` and `b` from `t`, the mappings in `d1` and `d2` mismatch, this means a prior different character already created a different mapping, hence they are not isomorphic, and we return `false`.

If the mappings agree for every pair of characters (implying consistent mapping), we update both arrays `d1` and `d2` marking the current position with the iteration index. If we complete the traversal without conflicts, the strings are isomorphic, indicated by returning `true`.

The idea is based on the concept that two isomorphic strings must have characters appear in the same order; hence their last occurrence should be at corresponding positions in both strings. This method allows us to verify that condition efficiently.

Solution Approach

The solution employs two arrays `d1` and `d2` of size 256 each (since the ASCII character set size is 256). These arrays are used to record the last seen positions of characters from `s` and `t` respectively. Another perspective is to consider `d1` and `d2` as mappings that keep track of positions rather than characters. This is subtly different from traditional mappings of characters to characters and avoids issues with checking for unique mappings back and forth.

Here is the step-by-step implementation breakdown:

1. We initialize two arrays `d1` and `d2` with zeros, assuming that a character's ASCII value is a valid index. This suits our purpose since characters that have not been seen are initialized with 0 by default.
2. We iterate over the strings `s` and `t` together. In Python, the `zip` function is useful for this as it bundles the two iterables into a single iterable each pair of characters can be accessed together in the loop.
3. During each iteration, we convert the characters `a` and `b` from `s` and `t` into their respective ASCII values using Python's `ord()` function.
4. We compare the current values at indices `a` and `b` in arrays `d1` and `d2`. If `d1[a]` is not equal to `d2[b]`, it implies that either `a` was previously associated with a different character than `b` or `b` with a different character than `a`, which means that the characters `a` and `b` are part of inconsistent mappings—violation of isomorphic property—hence, we return `false`.
5. If the comparison does not lead to a discrepancy, we need to update `d1[a]` and `d2[b]` with the current position of iteration, indicated by `i` (start counting from 1 to correctly handle the zeroth case). This step essentially marks the new last occurrence of each character pair as we proceed through the strings.
6. If all character pairs pass the condition checks and we successfully reach the end of the iteration, it suggests that all mappings were consistent. Hence, we return `true`.

By utilizing arrays and checking indices, we avoid the need for complicated hash table operations, which can be more expensive in terms of memory and time.

Here's the implementation of the above algorithm:

```python
class Solution:
    def isIsomorphic(self, s: str, t: str) -> bool:
        d1, d2 = [0] * 256, [0] * 256
        for i, (a, b) in enumerate(zip(s, t), 1):
            a, b = ord(a), ord(b)
            if d1[a] != d2[b]:
                return False
            d1[a] = d2[b] = i
        return True
```

In this approach, we use array indices to represent characters and array values to represent the last positions where characters appeared, comparing the evolution of these positions as a way to check for isomorphism.

Example Walkthrough

Let's use a small example to illustrate the solution approach. Suppose we have two strings `s = "paper"` and `t = "title"`. We want to determine if they are isomorphic.

1. We initialize two empty arrays `d1` and `d2` of size 256 to store last seen positions of characters from `s` and `t` respectively, starting with all zeros.
2. Iterating over the strings `s` and `t` together, we start with the first characters `p` from `s` and `t` from `t`. We convert them to their ASCII values and get `a = ord('p')` and `b = ord('t')`.
3. We check the current values at `d1[a]` and `d2[b]`. Since both are zero (indicating that we haven't encountered these characters before), we continue and update `d1[a]` and `d2[b]` with 1. Now `d1[ord('p')]` and `d2[ord('t')]` both hold the value 1.
4. Moving to the next characters, `a` from `s` and `i` from `t`, we again convert them to their ASCII values and get `a = ord('a')` and `b = ord('i')`. As before, `d1[ord('a')]` and `d2[ord('i')]` are both zero, so we update them to 2.
5. The following pairs are `p` and `t` again, and since we already have `d1[ord('p')] = d2[ord('t')] = 1` from the previous steps, this reaffirms the mapping, and we update these indices to 3 now.
6. Next pairs are `e` from `s` and `l` from `t`. They yield `a = ord('e')` and `b = ord('l')`. Finding both `d1[ord('e')]` and `d2[ord('l')]` at zero, we update their indices to 4.
7. The last characters are `r` from `s` and `e` from `t`, converting to `a = ord('r')` and `b = ord('e')`. As `d1[ord('r')]` and `d2[ord('e')]` are both zero, we update them to 5.
8. Now that we've reached the end of both strings without finding any discrepancies in the mapping, this means our mapping procedure remained consistent throughout. Thus, we conclude that `s = "paper"` and `t = "title"` are isomorphic and return `true`.

The solution confirms that corresponding characters in both strings appear in the same order and can be mapped one-to-one, fulfilling the condition for the strings to be isomorphic.

Python Solution

```python
class Solution:
    def isIsomorphic(self, s: str, t: str) -> bool:
        # Create two arrays to store the last seen positions of characters
        # from strings 's' and 't'.
        last_seen_s, last_seen_t = [0] * 256, [0] * 256

        # Iterate over the characters of the strings 's' and 't' simultaneously.
        for index, (char_s, char_t) in enumerate(zip(s, t), 1):  # Starting from 1
            # Convert the characters to their ASCII values
            ascii_s, ascii_t = ord(char_s), ord(char_t)

            # Check if the last seen positions for both characters are different.
            # If they are, then strings 's' and 't' are not isomorphic.
            if last_seen_s[ascii_s] != last_seen_t[ascii_t]:
                return False

            # Update the last seen positions for 'char_s', 'char_t' with the
            # current index which represents their new last seen position.
            last_seen_s[ascii_s] = last_seen_t[ascii_t] = index

        # If we have not found any characters with different last seen positions
        # till the end of both strings, then the strings are isomorphic.
        return True

```

Java Solution

```java
class Solution {

    // Method to check if two strings are isomorphic.
    // Two strings are isomorphic if the characters in string 's' can be replaced to get string 't'.
    public boolean isIsomorphic(String s, String t) {

        // Create two arrays to store the last seen positions of characters
        int[] lastSeenPositionInS = new int[256]; // Assuming extended ASCII
        int[] lastSeenPositionInT = new int[256]; // Assuming extended ASCII

        // Length of the input strings
        int length = s.length();

        // Iterate through each character in the strings
        for (int i = 0; i < length; ++i) {
            // Get the current characters from each string
            char charFromS = s.charAt(i);
            char charFromT = t.charAt(i);

            // If the last seen position of the respective characters
            // in the two strings are not the same, then they are not isomorphic
            if (lastSeenPositionInS[charFromS] != lastSeenPositionInT[charFromT]) {
                return false; // Not isomorphic
            }

            // Update the last seen position of the characters
            // i + 1 is used because default value in int arrays is 0,
            // and we are using the index as a check (can't use 0 as it is the default)
            lastSeenPositionInS[charFromS] = i + 1;
            lastSeenPositionInT[charFromT] = i + 1;
        }

        // If all characters in 's' can be replaced to get 't', return true, as the strings are isomorphic
        return true;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    bool isIsomorphic(string s, string t) {
        // Create mappings for characters in 's' and 't'
        int mappingS[256] = {0}; // Initialize to zero for all characters
        int mappingT[256] = {0}; // Initialize to zero for all characters
        int length = s.size();    // Get the size of the strings

        // Loop through each character in both strings
        for (int i = 0; i < length; ++i) {
            char charS = s[i]; // Character from string 's'
            char charT = t[i]; // Character from string 't'

            // Check if the current mapping does not match
            if (mappingS[charS] != mappingT[charT]) {
                return false; // If mappings don't match, strings are not isomorphic
            }

            // Update the mappings for the current characters
            // We use i + 1 because the default value is 0
            // and we want to differentiate between uninitialized and first character (at 0 index)
            mappingS[charS] = mappingT[charT] = i + 1;
        }

        // If all mappings matched, strings are isomorphic
        return true;
    }
};

```

Typescript Solution

```typescript
// Function to check whether two strings 's' and 't' are isomorphic.
// Two strings are isomorphic if the characters in 's' can be replaced to get 't'.
function isIsomorphic(source: string, target: string): boolean {
    // Create two arrays to store the last seen positions of characters in 'source' and 'target'.
    const sourceLastSeen: number[] = new Array(256).fill(0);
    const targetLastSeen: number[] = new Array(256).fill(0);

    // Iterate over each character in the strings.
    for (let i = 0; i < source.length; ++i) {
        // Get the character code for current characters in 'source' and 'target'.
        const sourceCharCode = source.charCodeAt(i);
        const targetCharCode = target.charCodeAt(i);

        // Check if the last seen position for both characters is the same.
        // If they differ, the strings are not isomorphic.
        if (sourceLastSeen[sourceCharCode] !== targetLastSeen[targetCharCode]) {
            return false;
        }

        // Update the last seen position for both characters to the current position + 1
        // The +1 ensures that the default value of 0 is not considered a position.
        sourceLastSeen[sourceCharCode] = i + 1;
        targetLastSeen[targetCharCode] = i + 1;
    }

    // If we complete the iteration without returning false, the strings are isomorphic.
    return true;
}

```

Time and Space Complexity

The time complexity of the code is `O(n)`. This is because the code iterates through all the characters of the strings `s` and `t` once, where `n` is the length of the string `s`. Each iteration involves constant time operations like accessing and updating the value in arrays `d1` and `d2`, and comparing the values.

The space complexity of the code is `O(C)`, where `C` is the constant size of the character set used in the problem. In this case, `C = 256`, which covers all the extended ASCII characters. The space is taken by two arrays `d1` and `d2`, each of which has 256 elements to accommodate all possible characters in the strings. Since the size of the character set does not scale with the size of the input, it is considered a constant space complexity.