205. Isomorphic Strings

Given two strings `s` and `t`, *determine if they are isomorphic*.

Two strings `s` and `t` are isomorphic if the characters in `s` can be replaced to get `t`.

All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character, but a character may map to itself.

Example 1:

```
Input: s = "egg", t = "add"
Output: true
```

Example 2:

```
Input: s = "foo", t = "bar"
Output: false
```

Example 3:

```
Input: s = "paper", t = "title"
Output: true
```

Constraints:

- `1 <= s.length <= 5 * 10<sup>4</sup>`
- `t.length == s.length`
- `s` and `t` consist of any valid ascii character.

```python
def isIsomorphic(self, s: str, t: str) -> bool:
        if len(s)!=len(t):
            return False

        map1 = {}
        used = {}
        for i in range(len(s)):
            ch1 = s[i]
            ch2 = t[i]
            if ch1 in map1:
                if map1[ch1]!=ch2:
```

```
                    return False
            else:
                if ch2 in used:
                    return False
                else:
                    map1[ch1] = ch2
                    used[ch2] = True
        return True
```

Approach 1: Character Mapping with Dictionary

Intuition

The first solution is based on the approach indicated in the problem statement itself. We will process both of the strings from left to right. At each step, we take one character at a time from the two strings and compare them. There are three cases we need to handle here:

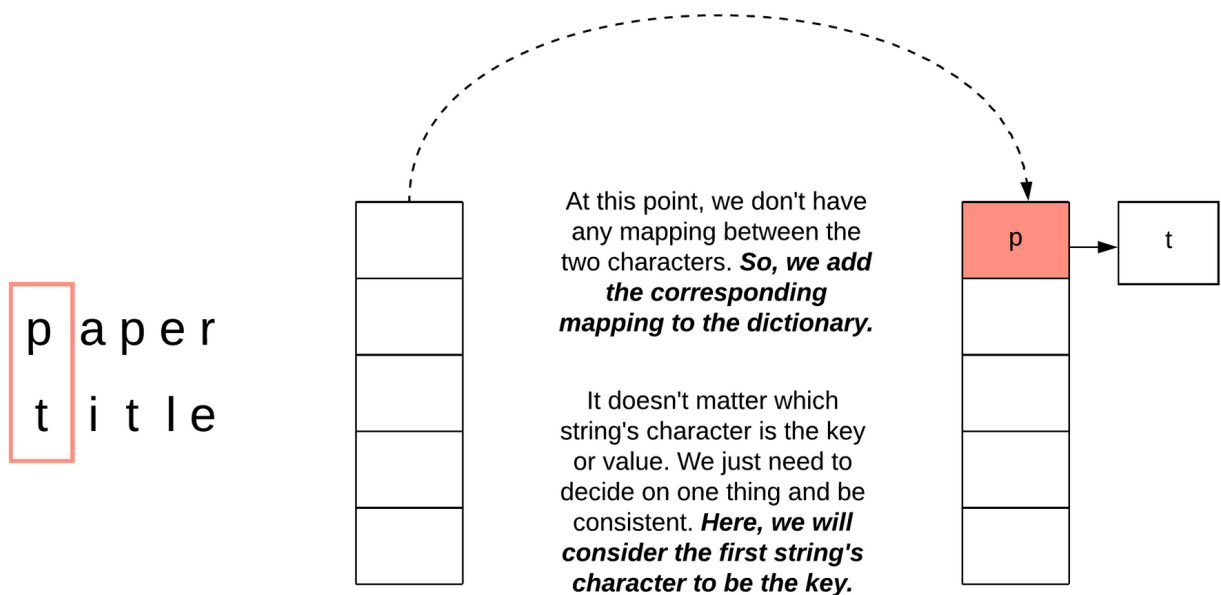1. **If the characters don't have a mapping, we add one in the dictionary and move on.**

Figure 1. The first encounter for a new character in both strings which are not yet mapped.

2. **The characters already have a mapping in the dictionary. If that is the case, then we're good to go.**

At this point, we see that there is a mapping between the two characters in the dicionary.

This is in compliance with our isomorphic requirement. Hence, we simply move on to the next character in both the strings.
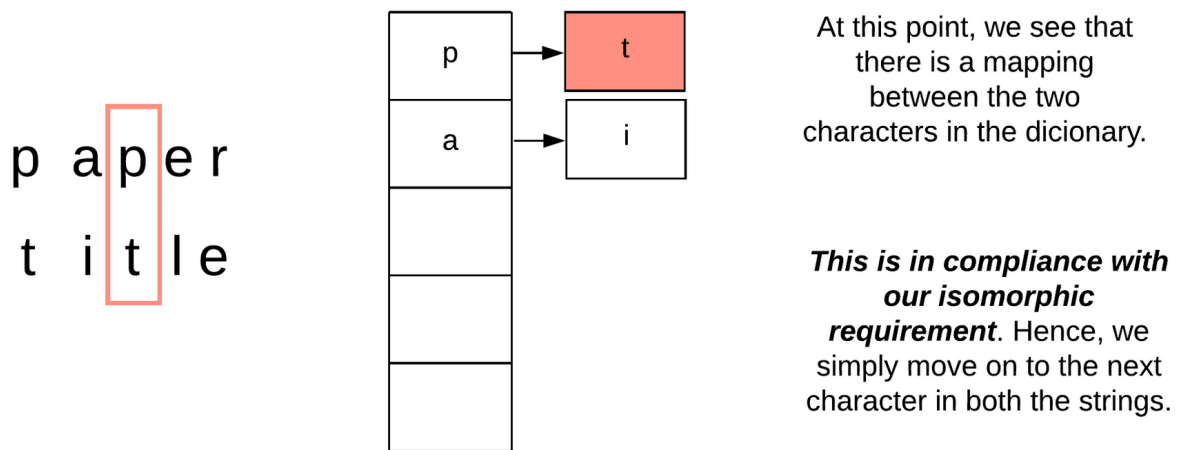
Figure 2. Example for when we already had a mapping between the corresponding characters.

3. *The final case is when a mapping already exists for one of the characters but it doesn't map to the other character at hand. In this case, we can safely conclude that the given strings are not isomorphic and we can return.*

If at this point you're ready to move on to the algorithm, take a step back and think about the correctness of this solution. The above three cases only care about **one-way-mapping** *i.e. mapping characters from the first string to the second one only. Don't we need the mapping from the other side as well?*
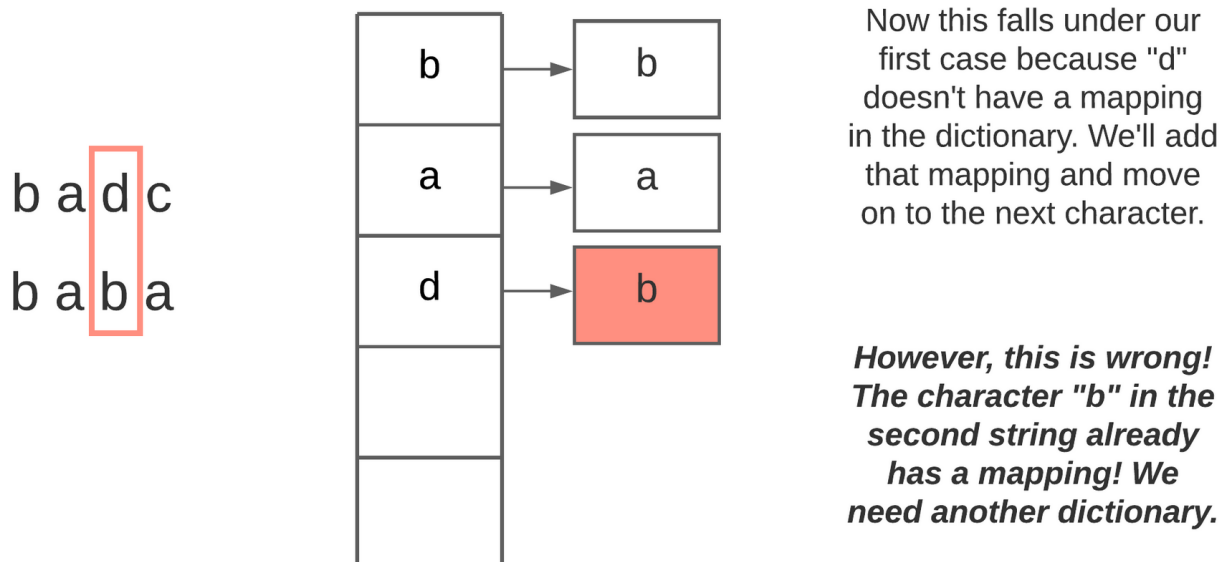

Now this falls under our first case because "d" doesn't have a mapping in the dictionary. We'll add that mapping and move on to the next character.

However, this is wrong! The character "b" in the second string already has a mapping! We need another dictionary.

Figure 3. Example for when a single dictionary implementation breaks.

We will need two dictionaries instead of one since we need `one-to-one` *mapping from the string* `s` *to string* `t` *and vice versa. Let's look at the algorithm to see the modified cases.*

Algorithm

1. *We define a dictionary* `mapping_s_t` *which will be used to map characters in string* `s` *to characters in string* `t` *and another dictionary* `mapping_t_s` *which will be used to map characters in string* `t` *to characters in string* `s`.

2. *Next, we iterate over the two strings one character at a time.*

3. **Let's assume the character in the first string is** `c1` **and the corresponding character in the second string is** `c2`.

    1. If `c1` does not have a mapping in `mapping_s_t` and `c2` does not have a mapping in `mapping_t_s`, we add the corresponding mappings in both the dictionaries and move on to the next character.

    2. At this point, we expect both the character mappings to exist in the dictionaries and their values should be `mapping_s_t[c1] = c2` and `mapping_t_s[c2] = c1`. If either of these conditions fails (`c1` is not in the dictionary, `c2` is not in the dictionary, unexpected mapping), we return `false`.

4. *Return* `true` *once both the strings have been exhausted.*

Implementation

```python
def isIsomorphic(self, s: str, t: str) -> bool:

        mapping_s_t = {}
        mapping_t_s = {}

        for c1, c2 in zip(s, t):

            # Case 1: No mapping exists in either of the dictionaries
            if (c1 not in mapping_s_t) and (c2 not in mapping_t_s):
                mapping_s_t[c1] = c2
                mapping_t_s[c2] = c1

            # Case 2: Ether mapping doesn't exist in one of the
dictionaries or Mapping exists and
            # it doesn't match in either of the dictionaries or both
            elif mapping_s_t.get(c1) != c2 or mapping_t_s.get(c2) != c1:
                return False

        return True
```

Complexity Analysis

Here N is the length of each string (if the strings are not the same length, then they cannot be isomorphic).

- Time Complexity: $O(N)$. We process each character in both the strings exactly once to determine if the strings are isomorphic.
- Space Complexity: $O(1)$ since the size of the ASCII character set is fixed and the keys in our dictionary are all valid ASCII characters according to the problem statement.

Approach 2: First occurence transformation

Intuition

This approach is based on the idea that the two given strings, if isomorphic, will in some way be exactly the same. If we have two isomorphic strings, we can replace the characters in the first string with the corresponding mapped characters to get the second string. The idea we explore here is the following:

> Is there any string transformation we can apply to both the strings such that to check for isomorphism, we simply check if their modified versions are **exactly** the same?

One can come up with various such transformations giving us different variations of this solution. We will stick with one such transformations for the official solution.

> For each character in the given string, we replace it with the index of that character's first occurence in the string.

For a string like `paper`, the transformed string will be `01034`. The character `p` occurs first at the index `0`; so we replace future occurrences of `p` with the index `0`. Similar modifications are made for the other characters. Now let's look at `title`. The transformed string would be `01034` which is the same as that for `paper`. This confirms the isomorphic nature of both the strings.

> However, we should be mindful of transformations that use both one and two-digit numbers. Under these circumstances, the transformed strings can be misinterpreted.

For example, `stenographics` and `logarithmsxox` both transform to `123456789110`, yet they are not isomorphic. So what went wrong? Well, the first 10 digits of `stenographics` are unique and the same is true for `logarithmsxox`, so they should be isomorphic up to `0123456789`. However the 11th and 12th characters of `stenographics` map to `11` and `0` while the 11th and 12th characters of `logarithmsxox` map to `1` and `10`. To avoid confusing `11 0` with `1 10` we can add a delimiter to help differentiate the transformed digits. Thus, by adding spaces we obtain `stenographics = 0 1 2 3 4 5 6 7 8 9 10 11 0` and `logarithmsxox = 0 1 2 3 4 5 6 7 8 9 10 1 10`. As a side note, this issue can also be resolved by comparing arrays of the transformed digits instead of using strings.

Algorithm

1. Define a function called `transform` that takes a string as an input and returns a new string with modifications as explained in the intuition section.
   1. We maintain a dictionary to store the character to index mapping for the given string.
   2. For each character, we look up the mapping in the dictionary. If there is a mapping, that means this character already has its first occurrence recorded and we simply use the first occurrence's index in the new string. Otherwise, we use the current index for the first occurrence.
2. We find the transformed strings for both of our input strings. Let's say the transformed strings are `s1` and `s2` respectively.
3. If `s1 == s2`, that implies the two input strings are isomorphic. Otherwise, they're not.

Implementation

```python
def transformString(self, s: str) -> str:
        index_mapping = {}
        new_str = []

        for i, c in enumerate(s):
            if c not in index_mapping:
                index_mapping[c] = i
            new_str.append(str(index_mapping[c]))

        return " ".join(new_str)

    def isIsomorphic(self, s: str, t: str) -> bool:
        return self.transformString(s) == self.transformString(t)
```

Complexity Analysis

Here N is the length of each string (if the strings are not the same length, they cannot be isomorphic).

- Time Complexity: $O(N)$. We process each character in both the strings exactly once to determine if they are isomorphic.
- Space Complexity: $O(N)$. We form two new strings returned by our transformation function. The size of ASCII character set is fixed and the keys in our dictionary are valid ASCII characters only. So the size of the map in the transform function doesn't contribute to the space complexity