1061. Lexicographically Smallest Equivalent String

Medium Union Find String

Leetcode Link

Problem Description

In this problem, you're given three strings `s1`, `s2`, and `baseStr`, all of which are of the same length. Characters at corresponding positions in `s1` and `s2` are considered equivalent, which means `s1[i]` is equivalent to `s2[i]`. This equivalence relationship is reflexive, symmetric, and transitive; that means if `a` is equivalent to `b`, then `b` is equivalent to `a` (symmetry), and if `a` is equivalent to `b`, `b` is equivalent to `c`, then `a` is equivalent to `c` (transitivity). Using these two equivalences, your goal is to transform `baseStr` into its lexicographically smallest equivalent string.

An example provided is with `s1 = "abc"` and `s2 = "cde"`, which means `a == c`, `b == d`, and `c == e`. For the `baseStr = "eed"`, both `"acd"` and `"aab"` are equivalent strings, but `"aab"` is the lexicographically smaller one.

Intuition

To find the lexicographically smallest string equivalent to `baseStr`, we can leverage a classic data structure known as the Union-Find or Disjoint Set Union (DSU). This structure helps efficiently handle equivalence relationships by finding and uniting equivalent components.

1. Initialize a disjoint set where each character initially belongs to its own set, which is indexed from 0 to 25 (mapping to 'a' through 'z').
2. Iterate over `s1` and `s2` simultaneously and, using the `find()` function, locate the root parent of the characters at the current indices in the Union-Find. If these roots are different, we unite the sets by making the root with a smaller index the parent of the root with a larger index. This step builds up the equivalence classes.
3. For each character in `baseStr`, replace it with the smallest equivalent character in its equivalence class. We find the smallest equivalent character by finding the root parent of the set the character belongs to.
4. After processing all characters of `baseStr`, the resulting string is the lexicographically smallest equivalent string.

The provided solution code follows the above approach using Union-Find to efficiently transform `baseStr` into its lexicographically smallest form respecting the equivalency relations defined by `s1` and `s2`.

Solution Approach

The solution uses the Union-Find algorithm to group equivalent characters together. Each character maps to an integer from 0 to 25, corresponding to 'a' through 'z'. The `find` function and an array `p` are used to manage the parent relationships of these integers, which is key to understanding the connected components (equivalence classes) of characters.

The implementation steps are as follows:

1. **Parent Array Initialization**: An array `p` of size 26 is initialized, where each index represents a unique character from 'a' to 'z'. Initially, each character is its own parent: `p[i] = i`.

```
1 p = list(range(26))
```

2. **Find Function**: The `find` function takes an integer `x` and finds the root parent in the Union-Find structure. This is done by recursively finding the parent until `p[x] == x`. Upon finding the root parent, path compression is performed by setting `p[x]` to the root parent to flatten the structure, making future queries faster.

```
1 def find(x):
2     if p[x] != x:
3         p[x] = find(p[x])
4     return p[x]
```

3. **Union Steps**: The main loop iterates over the indices of `s1` and `s2`, and for each pair of characters, it calculates their corresponding integer representations. Using the `find` function, it locates the root parents of each character's set. To maintain lexicographical order, it unions the sets by setting the parent of the higher index to the parent of the lower index.

```
1 for i in range(len(s1)):
2     a, b = ord(s1[i]) - ord('a'), ord(s2[i]) - ord('a')
3     pa, pb = find(a), find(b)
4     if pa < pb:
5         p[pb] = pa
6     else:
7         p[pa] = pb
```

4. **Building the Result String**: Lastly, for each character in `baseStr`, it is replaced by the lexicographically smallest equivalent character. This is achieved by finding the root parent of each character's set and converting it back to the character representation.

```
1 res = []
2 for a in baseStr:
3     a = ord(a) - ord('a')
4     res.append(chr(find(a) + ord('a')))
5 return ''.join(res)
```

By following these steps, the solution re-encodes `baseStr` into its smallest lexicographical form according to the equivalences defined by `s1` and `s2`, and returns the result as a new string.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Suppose we have the following input:

- `s1 = "ab"`
- `s2 = "bc"`
- `baseStr = "adb"`

The mapping arising from `s1` and `s2` means `a == b` and `b == c`. Since the equivalence relationship is transitive, we can also deduce that `a == c`.

Using the Union-Find algorithm, we'll follow the steps to find the lexicographically smallest equivalent string of `baseStr`.

1. **Parent Array Initialization**: We initialize the array `p` where `p[i] = i` for `i = 0` to `25`. For simplicity, assume the array is indexed starting at 0 for 'a', 1 for 'b', etc.

```
p = [0, 1, 2, ... 25]
```

2. **Find Function**: We define a `find` function that will be used to find the root parent of a character's set in our Union-Find structure, ensuring that we apply path compression along the way.
3. **Union Steps**: We iterate over `s1` and `s2` to build the equivalency classes:

   For `i = 0` (considering `a` and `b`):

   - `a` maps to `b`, and `b` maps to `b`.
   - `find(0)` returns `0`, and `find(1)` returns `1`.
   - Since `0 < 1`, we set `p[1]` to `0`.

   Now `p` looks like this: `[0, 0, 2, ... 25]`.

   For `i = 1` (considering `b` and `c`):

   - `b` maps to `b`, and `c` maps to `c`.
   - `find(1)` returns `0` (after path compression), and `find(2)` returns `2`.
   - Since `0 < 2`, we set `p[2]` to `0`.

   After this step, our `p` array reflects the transitive closure: `[0, 0, 0, ... 25]`.
4. **Building the Result String**: We construct the lexicographically smallest equivalent string for `baseStr`, "adb":

   - For `a` (`0`), `find(0)` returns `0`, so `a` remains `a`.
   - For `d` (`3`), `find(3)` returns `3`, so `d` remains `d`.
   - For `b` (`1`), `find(1)` now returns `0` due to path compression, so `b` is replaced by `a`.

   After this process, `baseStr` is transformed into `"aad"`.

As a result, the smallest equivalent string we can form from the base string "adb" following the equivalence relationships defined by `s1` and `s2` is "aad".

Python Solution

```
1  class Solution:
2      def smallestEquivalentString(self, s1: str, s2: str, baseStr: str) -> str:
3          # Initialize parent array for union-find structure to point
4          # each element to itself
5          parent = list(range(26))
6
7          # Find function uses path compression for finding the
8          # representative of a set.
9          def find(x):
10             if parent[x] != x:
11                 parent[x] = find(parent[x])
12             return parent[x]
13
14         # Merge the sets of characters in strings s1 and s2
15         for i in range(len(s1)):
16             parent_s1, parent_s2 = find(ord(s1[i]) - ord('a')), find(ord(s2[i]) - ord('a'))
17             parent_s1, parent_s2 = find(char_s1), find(char_s2)
18             # Link the sets by rank (in this case, by smallest representative).
19             if parent_s1 < parent_s2:
20                 parent[parent_s2] = parent_s1
21             else:
22                 parent[parent_s1] = parent_s2
23
24         # Build the resulting equivalent string based on the base string
25         # by replacing each character with its smallest equivalent.
26         result = []
27         for char in base_str:
28             char_index = ord(char) - ord('a')
29             result.append(chr(find(char_index) + ord('a')))
30
31         # Join and return the computed characters as a single string.
32         return ''.join(result)
33
```

Java Solution

```
1  class Solution {
2      // Parent array representing the disjoint set (union-find structure)
3      private int[] parent;
4
5      /**
6       * Generates the smallest equivalent string based on mapping rules.
7       *
8       * @param s1      - first input string with mapping rules
9       * @param s2      - second input string with mapping rules
10      * @param baseStr - the base string to be transformed
11      * @return The smallest lexicographical string after applying the rules from s1 and s2
12      */
13     public String smallestEquivalentString(String s1, String s2, String baseStr) {
14         // Initialize the parent array for 26 characters
15         parent = new int[26];
16         for (int i = 0; i < 26; ++i) {
17             parent[i] = i;
18         }
19
20         // Process the mappings in s1 and s2
21         for (int i = 0; i < s1.length(); ++i) {
22             // Convert the characters to zero-based indices
23             int index1 = s1.charAt(i) - 'a';
24             int index2 = s2.charAt(i) - 'a';
25
26             // Find the parents for both indices
27             int parent1 = find(index1);
28             int parent2 = find(index2);
29
30             // Union the parents by rank
31             if (parent1 < parent2) {
32                 parent[parent2] = parent1;
33             } else {
34                 parent[parent1] = parent2;
35             }
36         }
37
38         // Build the result string based on baseStr and union-find structure
39         StringBuilder result = new StringBuilder();
40         for (char ch : baseStr.toCharArray()) {
41             // Translate the base character according to the smallest parent character
42             char smallestEquivalentChar = (char) (find(ch - 'a') + 'a');
43             result.append(smallestEquivalentChar);
44         }
45         return result.toString();
46     }
47
48     /**
49      * Finds the representative of the set that element x is part of.
50      *
51      * @param x - an element for which to find the set representative
52      * @return The parent or the representative of the set
53      */
54     private int find(int x) {
55         // Path compression: Update the parent along the search path
56         if (parent[x] != x) {
57             parent[x] = find(parent[x]);
58         }
59         return parent[x];
60     }
61 }
62
```

C++ Solution

```
1  class Solution {
2  public:
3      vector<int> parent; // `parent` vector represents the parent of each character set
4
5      // Constructor initializes the `parent` vector to self indicating each character is its own parent
6      Solution() {
7          parent.resize(26);
8          iota(parent.begin(), parent.end(), 0); // 0, 1, 2, ..., 25
9      }
10
11     // Function to find the smallest lexicographical equivalent string
12     string smallestEquivalentString(string s1, string s2, string baseStr) {
13         // Union-Find Algorithm to merge the equivalence of characters in s1 and s2
14         for (int i = 0; i < s1.length(); ++i) {
15             int charIndex1 = s1[i] - 'a'; // convert character from 'a' to 'z' into index 0 to 25
16             int charIndex2 = s2[i] - 'a';
17             int parent1 = find(charIndex1), parent2 = find(charIndex2);
18             // If parents are different, unite by rank: attach the larger parent to the smaller one
19             if (parent1 < parent2) {
20                 parent[parent2] = parent1;
21             } else {
22                 parent[parent1] = parent2;
23             }
24         }
25
26         // Build the result by replacing each character in baseStr with its smallest equivalent
27         string result = "";
28         for (char c : baseStr) {
29             result += (find(c - 'a') + 'a');
30         }
31         return result;
32     }
33
34     // Function to find the representative of the set that 'x' belongs to
35     int find(int x) {
36         if (parent[x] != x) {
37             // Path compression: make every node on the path point directly to the root
38             parent[x] = find(parent[x]);
39         }
40         return parent[x];
41     }
42 };
43
```

Typescript Solution

```
1  // Global parent array to represent the parent of each character set
2  const parent: number[] = Array.from({ length: 26 }, (_, i) => i);
3
4  // Function to find the representative of the set that 'x' belongs to
5  function find(x: number): number {
6      // Path compression: make every node on the path point directly to the root
7      if (parent[x] !== x) {
8          parent[x] = find(parent[x]);
9      }
10     return parent[x];
11 }
12
13 // Function to merge the equivalence of characters in s1 and s2
14 function union(charIndex1: number, charIndex2: number): void {
15     const parent1 = find(charIndex1);
16     const parent2 = find(charIndex2);
17     // Union by rank: attach the larger parent to the smaller one
18     if (parent1 < parent2) {
19         parent[parent2] = parent1;
20     } else {
21         parent[parent1] = parent2;
22     }
23 }
24
25 // Function to find the smallest lexicographical equivalent string
26 function smallestEquivalentString(s1: string, s2: string, baseStr: string): string {
27     // Loop through the provided strings to perform the Union-Find algorithm
28     for (let i = 0; i < s1.length; ++i) {
29         const charIndex1 = s1.charCodeAt(i) - 'a'.charCodeAt(0); // convert character from 'a' to 'z' into index 0 to 25
30         const charIndex2 = s2.charCodeAt(i) - 'a'.charCodeAt(0);
31         union(charIndex1, charIndex2);
32     }
33
34     // Build the result by replacing each character in baseStr with its smallest equivalent
35     let result = "";
36     for (const c of baseStr) {
37         result += String.fromCharCode(find(c.charCodeAt(0) - 'a'.charCodeAt(0)) + 'a'.charCodeAt(0));
38     }
39     return result;
40 }
41
```

Time and Space Complexity

The given code represents a solution to find the smallest equivalent string based on the character mappings described by `s1` and `s2`, and then applies this mapping to `baseStr`.

Time Complexity:

The time complexity of the code is derived from the following operations:

- A loop that iterates through `s1` and `s2` strings (both are of equal length): `for i in range(len(s1))`.
- The `find` function, which is a path compression technique in the Union-Find algorithm. The amortized time complexity for each call to `find` is $O(\alpha(n))$, where $\alpha(n)$ is the Inverse Ackermann function. It is nearly constant and very slow-growing, basically considered a constant for practical purposes.
- Iterating through each character in `baseStr` and performing the `find` operation.

Since we run the `find` operation for each character in $s1/s2$ and then again for each character in `baseStr`, the total time complexity is $O(n * m * \alpha(n))$, where n is the length of $s1/s2$ and m is the length of `baseStr`. The term $\alpha(n)$ can usually be omitted when discussing practical time complexity, leading to $O(n * m)$.

Space Complexity:

The space complexity is determined by:

- The parent array `p` of fixed size 26, which corresponds to the 26 possible characters: $O(26)$ or $O(1)$ since it's a constant size.
- The `res` list that will eventually contain every character in `baseStr`: $O(m)$, where m is the length of `baseStr`.

Therefore, the space complexity is $O(m + 1)$ or simply $O(m)$ since the term $O(1)$ is negated when compared to non-constant space usage.

In conclusion, the time complexity of the provided code is $O(n * m)$ and the space complexity is $O(m)$.