1202. Smallest String With Swaps

Problem Description

The problem presents us with a string `s` and a list of `pairs`, where each `pair` contains two indices of the string `s`. We are allowed to swap characters at any of the given pair indices as many times as we wish. The objective is to determine the lexicographically smallest string that can be formed after making any number of swaps among the pairs.

Intuition

To arrive at the solution, we need to think about how the operations will affect the string `s`. Since any number of swaps can be made, if there is a way to swap between any two indices through a series of swaps, we can effectively consider those characters as part of the same group. This leads us to the concept of "connected components".

A connected component in this context is a set of indices where any index can reach any other index within the same set through the pairs given. Once we've identified these connected components, within each component, we can sort the characters lexicographically and place them back into their original positions. This is possible because swaps within a component can rearrange characters in any order.

To implement this, we use a union-find data structure (also known as the disjoint-set data structure). With union-find, we can efficiently join two indices together and identify which component an index belongs to.

The solution approach is as follows:

1. Initialize a union-find structure to keep track of the connected components. This structure will map an index to its root or representative index of its connected component.
2. Iterate through the pairs and perform the union operation for each pair, effectively joining two indices into the same connected component.
3. Organize the characters into groups by their root or representative index.
4. Sort each group of characters in reverse lexicographical order. This makes it easier to pop the smallest character when rebuilding the string.
5. Finally, build the result string by picking the smallest available character from each index's connected component. Since we sorted groups in reverse order, we pop characters from the end of each list, ensuring they are selected in increasing lexicographical order.

The provided function `find` is a helper function for the union-find structure that finds the root of an index, and during the process, applies path compression to keep the structure efficient by flattening the hierarchy (making each member of a component directly point to the root).

Solution Approach

The solution primarily utilizes the Union-Find algorithm and an efficient array-based approach to organize and sort the characters based on their connected components.

Here's the walkthrough of the solution approach:

1. **Initialization of Union-Find Structure**: We begin by creating an array `p` which will act as the parent array for the Union-Find algorithm. Each element's initial value is its own index, implying that it is its own root.

```
1 p = list(range(n)) # n is the length of string s
```

2. **Union Operations**: Iterate through the given `pairs` and perform the union operations. Instead of the traditional Union-Find with rank and path compression, the solution uses a simpler version that just assigns a new root to a component without considering the rank.

```
1 for a, b in pairs:
2     p[find(a)] = find(b)
```

The `find` function locates the root of an index and applies path compression by setting the `p[x]` entry directly to the root of `p[x]`.

3. **Grouping Characters By Component**: Once all pairs are processed, we need to group the characters according to their representative or root index. A dictionary `d` with default list values is used to append characters at the same group index.

```
1 d = defaultdict(list)
2 for i, c in enumerate(s):
3     d[find(i)].append(c)
```

4. **Sorting Groups**: Each list within dictionary `d` is sorted in reverse order. Sorting in reverse order enables us to pop elements from the end of the list later, which is faster than popping from the beginning.

```
1 for i in d.keys():
2     d[i].sort(reverse=True)
```

5. **Rebuilding the String**: The final string is formed by iterating over the original string's indices, finding the root of each index, and popping the smallest character from the corresponding group.

```
1 return "".join(d[find(i)].pop() for i in range(n))
```

The solution effectively groups indices of the string into connected components, sorts characters within each component, and then reconstructs the string by choosing the lexicographically smallest character available from each component for each position in the string. It's a smart application of the Union-Find algorithm and sorting to get the lexicographically smallest permutation satisfying the constraints.

Example Walkthrough

Let's consider a simple example to illustrate the solution approach with a string `s = "dcab"` and pairs `pairs = [[0, 3], [1, 2]]`.

1. **Initialization of Union-Find Structure**: We first initialize the parent array `p` for Union-Find. The string length is 4, so `p = [0, 1, 2, 3]`.
2. **Union Operations**: Next, we perform union operations with the given pairs.
   - For pair `[0, 3]`, we call `find(0)` and `find(3)` which return 0 and 3, and then we set `p[0]` to root of 3, which is 3. Now `p = [3, 1, 2, 3]`.
   - For pair `[1, 2]`, we call `find(1)` and `find(2)` which return 1 and 2, and then we set `p[1]` to root of 2, which is 2. Now `p = [3, 2, 2, 3]`.
3. **Grouping Characters by Component**: We group the characters by their roots. Resulting in `d = {3: ['a', 'b'], 2: ['d', 'c']}`.
4. **Sorting Groups**: Sort each group in reverse order:
   - For group at root 2, sort `['d', 'c']` to get `['b', 'a']`.
   - For group at root 3, sort `['a', 'c']` to get `['d', 'c']`. Now we have `d = {3: ['b', 'a'], 2: ['d', 'c']}`.
5. **Rebuilding the String**: We now rebuild the string by iterating over the original string's indices, finding the root of each index, and popping the smallest character from the corresponding group.
   - For index 0, its root is 3, so we pop from `d[3]` to get `'a'`. `d[3] = ['b']`.
   - For index 1, its root is 2, so we pop from `d[2]` to get `'d'`. `d[2] = ['c']`.
   - For index 2, its root is also 2, so we pop again from `d[2]` to get `'c'`. `d[2] = []`.
   - For index 3, its root is 3, so we pop from `d[3]` to get `'b'`. `d[3] = []`.

Finally, the rebuilt string is `"acbd"`, which is the lexicographically smallest string we can obtain through the allowed swaps.

The entire process utilizes the union-find algorithm to efficiently manage connected components and applies sorting to ensure the smallest characters are placed first. The resulting string is created by choosing the smallest character allowed by the swap operations at each index.

Python Solution

```
1 from collections import defaultdict
2
3 class Solution:
4     def smallestStringWithSwaps(self, s: str, pairs: List[List[int]]) -> str:
5         # Function to find the root of the set that element a belongs to
6         def find_root(a: int) -> int:
7             # Path compression: update parent to root for quicker access next time
8             if parent[a] != a:
9                 parent[a] = find_root(parent[a])
10             return parent[a]
11
12         # Initialize the number of characters in the string
13         n = len(s)
14         # Initially, every character is in its own set
15         parent = list(range(n))
16
17         # Union operation: merge sets that are paired
18         for a, b in pairs:
19             # Set the parent of a to be the root of set b, effectively merging the sets
20             parent[find_root(a)] = find_root(b)
21
22         # A dictionary to hold the characters by the root of the set they belong to
23         char_group = defaultdict(list)
24
25         # Group all characters by their root
26         for i, char in enumerate(s):
27             char_group[find_root(i)].append(char)
28
29         # Sort characters in each group in descending order to pop smallest char later
30         for chars in char_group.values():
31             chars.sort(reverse=True)
32
33         # Build the smallest string by popping the smallest available character from the group
34         output = [char_group[find_root(i)].pop() for i in range(n)]
35         # Join the characters to form the result string
36         return "".join(output)
```

Java Solution

```
1  class Solution {
2      // The parent array for the disjoint set (Union-Find) data structure
3      private int[] parent;
4
5      /**
6       * Generates the smallest string resulting from swapping indices in pairs
7       *
8       * @param s The original string
9       * @param pairs A list of index pairs for swapping
10      * @return The lexicographically smallest string after swaps
11      */
12     public String smallestStringWithSwaps(String s, List<List<Integer>> pairs) {
13         int length = s.length();
14         parent = new int[length];
15         List<Character>[] disjointSets = new List[length];
16
17         // Initialize parent pointers to themselves and disjoint sets
18         for (int i = 0; i < length; ++i) {
19             parent[i] = i;
20             disjointSets[i] = new ArrayList<>();
21         }
22
23         // Join sets using union-find with the given pairs
24         for (List<Integer> pair : pairs) {
25             int a = pair.get(0), b = pair.get(1);
26             parent[find(a)] = find(b);
27         }
28
29         // Distribute characters to corresponding sets
30         for (int i = 0; i < length; ++i) {
31             int j = find(i);
32             disjointSets[j].add(s.charAt(i));
33         }
34
35         // Sort each set in descending order (since we'll be removing from the end)
36         for (List<Character> set : disjointSets) {
37             Collections.sort(set, (a, b) -> b - a);
38         }
39
40         // Build the result by choosing the last element in each set
41         StringBuilder result = new StringBuilder();
42         for (int i = 0; i < length; ++i) {
43             int j = find(i);
44             result.append(disjointSets[j].remove(disjointSets[j].size() - 1));
45         }
46
47         // Return the resultant string
48         return result.toString();
49     }
50
51     /**
52      * Finds the root of the set a belongs to
53      *
54      * @param a The element to find the set of
55      * @return The root of the set containing x
56      */
57     private int find(int a) {
58         // Path compression for efficiency
59         if (parent[a] != a) {
60             parent[a] = find(parent[a]);
61         }
62         return parent[a];
63     }
64 }
```

C++ Solution

```
1  class Solution {
2  public:
3      // Function to generate the lexicographically smallest string possible by swapping indices given by pairs
4      string smallestStringWithSwaps(string s, vector<vector<int>>& pairs) {
5          int n = s.size();                   // Length of the input string
6          vector<int> parent(n, -1);          // Vector to keep track of the connected components
7          function<int(int)> find = [&](int x) -> int { // Lambda function to find and convert to represent distinct sets
8              return parent[x] < 0 ? x : parent[x] = find(parent[x]); // Path compression
9          };
10         function<void(int, int)> join = [&](int x, int y) { // Lambda function for joining the root of the set
11             parent[find(x)] = find(y);
12         };
13
14         // Lambda function that implements path compression for the disjoint set
15         for (int i = 0; i < n; ++i) {
16             if (parent[i] >= 0) {
17                 parent[i] = find(parent[i]);
18             }
19             return parent[i];
20         };
21
22         // Iterate over all pairs and perform the union operation to join the connected components
23         for (const auto& edge : pairs) {
24             int a = find(edge[0]), b = find(edge[1]);
25             parent[a] += b; // Merge the components of a with the component of b
26             parent[b] = a;
27         }
28
29         // Group characters by the root of their connected component
30         for (int i = 0; i < n; ++i) {
31             charGroups[find(i)].push_back(s[i]); // Add character to its representative's group
32         }
33
34         // Sort the characters within each group in reverse order (since we will later pop elements from the end)
35         for (auto& group : charGroups) {
36             sort(group.rbegin(), group.rend());
37         }
38
39         // Construct the lexicographically smallest string by picking the smallest available character from each group
40         for (int i = 0; i < n; ++i) {
41             int root = find(i); // Find the root of the connected characters for the current index
42             result.push_back(charGroups[root].back()); // Append the character to the result string
43             group.pop_back(); // Remove the used character from the group
44         }
45
46         return s; // Return the lexicographically smallest string after swaps
47     }
48 };
```

Typescript Solution

```
1  // Function to find the smallest lexicographical string that can be obtained by swapping any pairs of indices.
2  function smallestStringWithSwaps(s: string, pairs: number[][]): string {
3      const length = s.length;
4      // Parent array for Union-Find to keep track of connected components
5      const parent = Array.from({ length }, (_, index) => index);
6
7      // Function to find the root parent of a component using path compression
8      const find = (index: number): number => {
9          if (parent[index] !== index) {
10             parent[index] = find(parent[index]);
11         }
12         return parent[index];
13     };
14
15     // Array of arrays to store characters belonging to the same connected component
16     const components: string[][] = new Array(length).fill(0).map(() => []);
17
18     // Union-Find to connect components
19     for (const [a, b] of pairs) {
20         const rootA = find(a);
21         const rootB = find(b);
22         if (rootA !== rootB) {
23             parent[rootA] = rootB;
24         }
25     }
26
27     // Grouping the characters by their connected components
28     for (let i = 0; i < length; ++i) {
29         components[find(i)].push(s[i]);
30     }
31
32     // Sorting the characters in each component in descending order
33     // To access them in ascending order later using pop()
34     for (const component of components) {
35         component.sort((a, b) => b.charCodeAt(0) - a.charCodeAt(0));
36     }
37
38     // Array to build the final answer
39     const answer: string[] = [];
40     for (let i = 0; i < length; ++i) {
41         // Retrieve and remove the last character from the sorted component
42         answer.push(components[find(i)].pop()!);
43     }
44
45     // Join all characters to form the resulting string and return
46     return answer.join('');
47 }
48
```

Time and Space Complexity

Time Complexity

The time complexity of the code can be broken down into the following parts:

1. **Union-Find Initialization**: Initializing the parent list `p` with `n` elements where `n` is the length of string `s` takes O(n) time.
2. **Union-Find Path Compression**: For each pair in `pairs`, we potentially perform a *union-by-rank* and *path compression*. In the worst-case scenario, all elements can be connected, which ends up being nearly $O(\alpha(n))$ per operation where $\alpha(n)$ is the inverse Ackermann function. It is very slowly growing and is less than 5 for all practical values of `n`. Since there can be `m` pairs, this step takes at most $O(m * \alpha(n))$ time.
3. **Grouping letters by connected components**: Enumerating `s` and finding the root of each index takes O(n * alpha(n)) time. Appending characters to the corresponding lists in the dictionary `d` takes O(1) time per character, resulting in O(n) for this step.
4. **Sorting the letters**: Each list in the dictionary `d` is sorted in reverse order. If `k` is the maximum size of the lists to be sorted, this step takes O(k * log(k)) time for each connected component. Since in the worst case there can be n/k such components, the overall sorting complexity can be $O(n/k) * k * \log(k)) = O(n * \log(k))$.
5. **Reconstructing the result string**: Popping an element from each list in `d` and forming a new string takes O(n) time.

Summing these up, the worst-case time complexity of the algorithm is $O(n * \alpha(n) + n * \alpha(n) * \log(k) + n)$, where $\alpha(n)$ is a very slow-growing function and can be considered constant for all practical purposes, and $\log(k) <= \log(n)$, we can simplify this to $O(n * \log(n) + m)$.

Space Complexity

The space complexity can be broken down into the following:

1. **Parent array** `p`: Uses O(n) space.
2. **Dictionary** `d`: Contains all most `n` lists. In the worst case, each list has one character, so the total space taken by `d` is O(n).
3. **Auxiliary space for sorting**: The in-place sorting algorithm used by Python's `.sort()`, the space complexity is O(1). However, if we consider the space for the call stack due to recursive sorting algorithms in some implementations, this could give an additional space of O(log(k)) per connected component which sums up to O(n) in total.

Hence, the total space complexity is $O(n)$.