1202. Smallest String With Swaps Medium **Depth-First Search Breadth-First Search Union Find** 

# **Problem Description**

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

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

Array

Hash Table

String

Sorting

**Leetcode Link** 

characters in any order.

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

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.

point to the root).

**Solution Approach** 

based on their connected components.

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

the end of the list later, which is faster than popping from the beginning.

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

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

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.

The solution primarily utilizes the Union-Find algorithm and an efficient array-based approach to organize and sort the characters

## 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: 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

representative or root index. A dictionary d with default list values is used to append characters at the same group index.

2. Union Operations: Iterate through the given pairs and perform the union operations. Instead of the traditional Union-Find with

p[x]. 3. Grouping Characters By Component: Once all pairs are processed, we need to group the characters according to their

1 d = defaultdict(list)

1 for i in d.keys():

Example Walkthrough

[3, 1, 2, 3].

4. **Sorting Groups**: Sort each group in reverse order:

order to maintain the lexicographical order.

For group at root 2, sort ['a', 'b'] to get ['b', 'a'].

For index 3, its root is 3, so we pop from d[3] to get 'd' (d[3] = []).

2, 2, 3].

'c']}.

constraints.

2 for i, c in enumerate(s):
3 d[find(i)].append(c)

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.

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

4. Sorting Groups: Each list within dictionary d is sorted in reverse order. Sorting in reverse order enables us to pop elements from

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. Grouping Characters By Component: We group the characters by their roots. Resulting in d = {2: ['a', 'b'], 3: ['d',

For group at root 3, sort ['d', 'c'] to get ['d', 'c']. Now we have d = {2: ['b', 'a'], 3: ['d', 'c']}.

∘ 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,

1. Initialization of Union-Find Structure: We first initialize the parent array p for Union-Find. The string length n is 4, so p = [0, 1,

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

operations at each index.

**Python Solution** 

10

14

15

16

17

20

21

23

24

33

35

36

37

38

39

10

11

12

13

14

16

22

23

24

25

26

27

30

31

32

33

34

35

36

37

38

39

40

41

42

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

6

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

40

41

42

43

44

45

46

48

47 }

**}**;

**}**;

Typescript Solution

from collections import defaultdict

if parent[x] != x:

return parent[x]

parent = list(range(n))

char\_group = defaultdict(list)

chars.sort(reverse=True)

# the current position belongs to

for a, b in pairs:

return output

private int[] parent;

\* @param s The original string

int length = s.length();

for (var pair : pairs) {

parent[find(a)] = find(b);

char[] chars = s.toCharArray();

for (var set : disjointSets) {

set.sort((a, b) -> b - a);

for (int i = 0; i < length; ++i) {</pre>

for (auto& group : charGroups) {

for (int i = 0; i < n; ++i) {

const lengthOfString = s.length;

**if** (parent[x] !== x) {

// Union-Find to connect components

const rootA = findRoot(a);

const rootB = findRoot(b);

parent[rootA] = rootB;

for (const component of components) {

for (let i = 0; i < lengthOfString; ++i) {</pre>

return sortedString.join('');

Time and Space Complexity

for (let i = 0; i < lengthOfString; ++i) {</pre>

components[findRoot(i)].push(s[i]);

// Grouping the characters by their connected components

// to access them in ascending order later using pop()

sortedString.push(components[findRoot(i)].pop()!);

// Join all characters to form the resulting string and return

// Sorting the characters in each component in descending order

// Retrieve and remove the last character from the sorted component

for (const [a, b] of pairs) {

if (rootA !== rootB) {

return parent[x];

const findRoot = (x: number): number => {

parent[x] = findRoot(parent[x]);

sort(group.rbegin(), group.rend());

for (int i = 0; i < length; ++i) {</pre>

parent = new int[length];

\* @param pairs A list of index pairs for swapping

\* @return The lexicographically smallest string after swaps

List<Character>[] disjointSets = new List[length];

// Join sets using union-find with the given pairs

int a = pair.get(0), b = pair.get(1);

// Distribute characters to corresponding sets

disjointSets[find(i)].add(chars[i]);

n = len(s)

parent[x] = find\_root(parent[x])

# Initially, every character is in its own set

# Union operation: merge sets that are paired

parent[find\_root(a)] = find\_root(b)

# Initialize the number of characters in the string

For index 0, its root is 3, so we pop from d[3] to get 'c' (d[3] = ['d']). For index 1, its root is 2, so we pop from d[2] to get 'a' (d[2] = ['b']). ∘ For index 2, its root is also 2, so we pop again from d[2] to get 'b' (d[2] = []).

The entire process utilizes the union-find algorithm to efficiently manage connected components and applies sorting to ensure the

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

smallest characters are placed first. The resulting string is created by choosing the smallest character allowed by the swap

5. Rebuilding the String: We now rebuild the string by iterating over each index remembering to pop from the end of the list in

class Solution: def smallestStringWithSwaps(self, s: str, pairs: List[List[int]]) -> str: # Function to find the root of the set that element x belongs to def find\_root(x: int) -> int:

# Path compression: update parent to root for quicker access next time

# Set the parent of set a to be the root of set b, effectively merging the sets

# Build the smallest string by popping the smallest available character from the group

# A dictionary to hold the characters by the root of the set they belong to

output = "".join(char\_group[find\_root(i)].pop() for i in range(n))

// The parent array for the disjoint set (Union-Find) data structure

\* Generates the smallest string resulting from swapping indices in pairs

public String smallestStringWithSwaps(String s, List<List<Integer>> pairs) {

// Sort each set in descending order (since we'll be removing from the end)

// Build the result by choosing the last element in each set

26 # Group all characters by their root 27 for i, c in enumerate(s): 28 char\_group[find\_root(i)].append(c) 29 # Sort characters in each group in descending order to pop smallest char later 30 for chars in char\_group.values(): 31

```
17
            // Initialize parent pointers to themselves and disjoint sets
            for (int i = 0; i < length; ++i) {</pre>
18
                parent[i] = i;
19
                disjointSets[i] = new ArrayList<>();
20
21
```

Java Solution

class Solution {

/\*\*

\*/

```
var set = disjointSets[find(i)];
43
               chars[i] = set.remove(set.size() - 1);
44
45
46
           // Return the resultant string
           return String.valueOf(chars);
47
48
49
50
       /**
51
        * Finds the root of the set x belongs to
52
53
        * @param x The element to find the set of
54
        * @return The root of the set containing x
55
        */
56
       private int find(int x) {
           // Path compression for efficiency
57
58
           if (parent[x] != x) {
               parent[x] = find(parent[x]);
59
60
           return parent[x];
61
62
63 }
64
C++ Solution
  1 class Solution {
  2 public:
         // Function to generate the lexicographically smallest string possible by swapping indices given by pairs
         string smallestStringWithSwaps(string s, vector<vector<int>>& pairs) {
             int n = s.size();
                                              // Length of the input string
                                              // Vector to keep track of the connected components
             vector<int> parent(n);
  6
             iota(parent.begin(), parent.end(), 0); // Initialize each node as its own parent to represent distinct sets
             vector<vector<char>> charGroups(n); // Each index can store characters belonging to the same connected component
  8
             function<int(int)> find; // Declaration of the 'find' lambda function for finding the root of the set
  9
 10
 11
             // Lambda function that implements path compression for the disjoint set
 12
             find = [\&](int x) \rightarrow int {
                 if (parent[x] != x) {
 13
                     parent[x] = find(parent[x]);
 14
 15
 16
                 return parent[x];
 17
             };
 18
 19
             // Iterate over all pairs and perform the union operation to join the connected components
 20
             for (const auto& edge : pairs) {
                 int a = find(edge[0]), b = find(edge[1]);
 21
 22
                 parent[a] = b; // Merge the component of a with the component of b
 23
 24
 25
             // Group characters by the root of their connected component
 26
             for (int i = 0; i < n; ++i) {
 27
                 charGroups[find(i)].push_back(s[i]); // Add character to its representative's group
 28
 29
 30
             // Sort the characters within each group in reverse order (since we will later pop elements from the end)
```

// Construct the lexicographically smallest string by picking the smallest available character from each group

auto& group = charGroups[find(i)]; // Find the group of connected characters for the current index

s[i] = group.back(); // Assign the smallest character to the current index

1 // Function to find the smallest lexicographical string that can be obtained by swapping any pairs of indices.

group.pop back(); // Remove the used character from the group

return s; // Return the lexicographically smallest string after swaps

function smallestStringWithSwaps(s: string, pairs: number[][]): string {

// Parent array for Union-Find to keep track of connected components

const parent = new Array(lengthOfString).fill(0).map((\_, index) => index);

// Function to find the root parent of a component using path compression

// Array of arrays to store characters belonging to the same connected component

const components: string[][] = new Array(lengthOfString).fill(0).map(() => []);

## component.sort((a, b) => b.charCodeAt(0) - a.charCodeAt(0)); 35 36 37 38 // Array to build the final answer const sortedString: string[] = []; 39

Time Complexity

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 0(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.

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

- 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 0(n + m \* alpha(n) + n \* alpha(n) + n \* log(k) + n). Since 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).

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

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

- **Space Complexity** 
  - 2. Dictionary d: Contains at 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: In 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

1. Parent array p: Uses O(n) space.

The space complexity can be broken down into the following:

- 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).