1081. Smallest Subsequence of Distinct Characters String ] **Monotonic Stack** Medium Greedy

# **Problem Description**

The problem requires us to find a specific type of subsequence from a given string s. A subsequence is a sequence that can be

derived from the original string by deleting some or no characters without changing the order of the remaining characters. The

goal is to determine the lexicographically smallest subsequence that contains every distinct character of s exactly once. • Lexicographically smallest: This means the sequence that would appear first if all possible subsequences were sorted in dictionary order.

Let's break it down:

- order of the remaining elements. • Contains all the distinct characters of s exactly once: The subsequence must have all unique characters that appear in s, and none of these characters should be repeated.
- This is like a puzzle where you need to choose characters, ensuring that you include each unique character at least once, but you can't choose a later character if it will cause an earlier unique character to never appear again in the remaining part of the string

To find the lexicographically smallest subsequence, we should take the smallest available character unless taking it would

prevent us from including another distinct character later on. We can use a stack to keep track of the characters in the current

subsequence as we iterate through the string. If we can add a smaller character and still have the opportunity to add the

• Subsequence: This is a sequence that can be derived from another sequence by deleting some or none of the elements without changing the

### characters currently in the stack later, we should do that to ensure our subsequence is the smallest possible lexicographically. The intuition to make this efficient is as follows:

Intuition

• Track the last occurrence of each character, so we know if we'll see a character again in the future. • Use a stack to maintain the characters of the current smallest subsequence. • Make use of a set to quickly check if a character is already in our stack (and hence, in our current smallest subsequence). • When we consider adding a new character, check if it is smaller than the last one on the stack, and if we'll encounter the last one on the stack again later (thanks to our tracking of the last occurrence). • If both conditions are true, we can safely pop the larger character off the stack and remove it from our set without fearing that we won't have it

- By following these steps, we can build the smallest subsequence as we iterate through the string s once.
- **Solution Approach** The solution uses a greedy algorithm with a stack to generate the lexicographically smallest subsequence. Here is a step-by-step
- explanation of how the code makes use of this approach: Track the Last Occurrence: We need to know the last position at which each character appears in the string. This is essential to decide whether we can drop a character from the stack for a smaller one. The last dictionary is used to store the last

index for every character using a dictionary comprehension:  $last = \{c: i \text{ for } i, c \text{ in enumerate}(s)\}$ .

our last occurrence tracking. We pop the character from stack and remove it from vis set.

in our subsequence.

As we iterate through the string, characters are pushed onto the stack if they can potentially be part of the lexicographically smallest subsequence. The stack, stk, is initialized as an empty list []. Set for Visited Characters: To ensure that each character is added only once to the subsequence, a set vis is used to track

**Stack for Current Smallest Subsequence:** A stack data structure is ideal for maintaining the current sequence of characters.

the characters that are already present in the stack. Iterate Over the String: We iterate over each character c and its index i in the string using for i, c in enumerate(s). If the character has already been visited, we continue to the next iteration with continue.

Character Comparison and Stack Pop: This is where the greedy algorithm comes into play. For the current character c, we

check if the stack is not empty and the char at the top of the stack is greater than c, and also if the character at the top of

the stack occurs later in the string (last[stk[-1]] > i). If all these conditions are true, it means we can pop the top of the

stack to make our subsequence lexicographically smaller and we are sure that this character will come again later, thanks to

Add the Current Character to the Stack and Visited Set: After making sure the characters on the stack are in the correct

lexicographical order and popping those that aren't, we can safely push the current character onto the stack and mark it as

visited by adding it to the vis set with vis.add(c). Build and Return the Result: At the end, the stack stk contains the lexicographically smallest subsequence. We join all the characters in the stack to form a string and return it with return "".join(stk).

By using a stack along with a set and a last occurrence tracking dictionary, the given Python code efficiently computes the

Let's consider a simple example to illustrate the solution approach with the string s = "cbacdcbc". We want to find the

lexicographically smallest subsequence that contains all distinct characters of the input string s exactly once.

lexicographically smallest subsequence which contains every distinct character exactly once.

o last occurrence of 'd' at index 4 So, last = {'c': 7, 'b': 6, 'a': 2, 'd': 4}.

i. On encountering the first 'c', it is not in vis, so we add 'c' to stk and vis.

ii. Next, 'b' is not in vis, so we add 'b' to stk and vis. stk now contains ['c', 'b'].

v. 'd' is encountered, not in vis, so we add it to stk. stk now contains ['a', 'c', 'd'].

vi. Next, we see another 'c'. It's already in vis, so we skip it.

viii. Lastly, 'c' comes again, but it is in vis, so we skip it.

the stack to get "acb", which is our final answer.

def smallestSubsequence(self, s: str) -> str:

# Result stack to build the smallest subsequence

# Iterate through the string's characters

visited.remove(stack.pop())

public String smallestSubsequence(String text) {

// Count array for each character 'a' through 'z'

// Iterate over each character in the input text

// Decrement the count of the current character

for index, character in enumerate(s):

if character in visited:

# that occurs later

stack.append(character)

visited.add(character)

int[] charCount = new int[26];

charCount[c - 'a']++;

for (char c : text.toCharArray()) {

for (char c : text.toCharArray()) {

result[++stackTop] = c;

visited[c - 'a'] = true;

string smallestSubsequence(string s) {

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

int lastIndex0f[26] = {0};

return String.valueOf(result, 0, stackTop + 1);

charCount[c - 'a']--;

if (!visited[c - 'a']) {

return "".join(stack)

# Set to keep track of characters already in the stack

# Ignore characters already added to the stack

# Dictionary to store the last occurrence index for each character

last\_occurrence = {character: index for index, character in enumerate(s)}

while stack and stack[-1] > character and last occurrence[stack[-1]] > index:

# Remove the character from visited when it is popped from the stack

# Add the current character to the stack and mark it as visited

// Fill the count array with the frequency of each character in the text

// If the character has not been visited, we need to include it in the result

// While the stack is not empty, the current character is smaller than the

// character at the stack's top and the character at the stack's top still

// occurs later in the text (i.e., the count is greater than 0), we pop

// Build the output string from the stack, which contains the smallest subsequence

int strSize = s.size(); // Determine the length of the input string.

// Array to store the last position (index) of each character in the string.

string result; // This will hold the smallest lexicographical subsequence.

const getIndex = (char: string): number => char.charCodeAt(0) - 'a'.charCodeAt(0);

// Ensure characters in stack are smallest possible and remove if not needed

// Last occurrence index of each character in the alphabet

// Update last occurrence index for each character in the string

const lastOccurrence: number[] = new Array(26).fill(0);

// Bitmask to track which characters are in the stack

// Skip if the character is already in the stack

const lastCharIndex = getIndex(stack.pop()!);

// Update the bitmask to include the current character

// Join the stack into a string and return it as the smallest subsequence

lastOccurrence[getIndex(char)] = index;

// Stack to hold the characters for the result

// Iterate over each character in the string

const charIndex = getIndex(char);

mask ^= 1 << lastCharIndex;</pre>

// Add the current character to the stack

# Iterate through the string's characters

visited.remove(stack.pop())

# Ignore characters already added to the stack

# Ensure characters in stack are in ascending order and

# remove any character that can be replaced with a lesser character

# Add the current character to the stack and mark it as visited

# Join the characters in the stack to form the smallest subsequence

while stack and stack[-1] > character and last occurrence[stack[-1]] > index:

# Remove the character from visited when it is popped from the stack

for index, character in enumerate(s):

if character in visited:

continue

# that occurs later

stack.append(character)

visited.add(character)

return "".join(stack)

Time and Space Complexity

if ((mask >> charIndex) & 1) {

[...s].forEach((char, index) => {

[...s].forEach((char. index) => {

const stack: string[] = [];

return;

stack.push(char);

return stack.join('');

visited = set()

mask |= 1 << charIndex;</pre>

});

let mask = 0;

lastIndexOf[s[i] - 'a'] = i; // Populate the array with the last index of each character.

// Visited array to track if a character is in the current result

# Join the characters in the stack to form the smallest subsequence

Solution Implementation

**Python** 

Java

class Solution {

class Solution:

stack = []

visited = set()

iv. 'c' comes again, we add it back since 'a' < 'c' and 'c' is not in vis. stk now contains ['a', 'c'].

- Track the Last Occurrence: First, we create a dictionary to track the last occurrence of each character: last occurrence of 'c' at index 7 last occurrence of 'b' at index 6
- Stack for Current Smallest Subsequence: We initialize an empty stack stk = [] to keep track of the subsequence characters.

Iterate Over the String: We iterate over each character in "cbacdcbc". Let's go through this process step by step:

**Set for Visited Characters**: We also have a set vis = {} to mark characters that we have already seen and added to the

### iii. 'a' is not in vis, so we add 'a' to stk. However, 'a' is smaller than 'b' and 'c', and both 'b' and 'c' will come later in the string. We pop 'b' and 'c' from the stack and remove them from vis. Then, add 'a' to stk and vis. stk now contains ['a'].

stack.

**Example Walkthrough** 

last occurrence of 'a' at index 2

Character Comparison and Stack Pop: Throughout the iteration, whenever we meet a smaller character, we check if we can pop the larger ones before it, as described above.

Add the Current Character to the Stack and Visited Set: We've been doing this in each iteration whenever necessary.

Build and Return the Result: At the end, stk contains the lexicographically smallest subsequence. We join the elements of

We successfully found the lexicographically smallest subsequence "acb" that contains every distinct character of s exactly once.

dictionary), we will pop 'd' from stk and remove it from vis, and add 'b' instead. stk now contains ['a', 'c', 'b'].

vii. 'b' comes next, is not in vis, and is less than 'd'. Since 'd' appears again later (we know from our last occurrence

- continue # Ensure characters in stack are in ascending order and # remove any character that can be replaced with a lesser character
- boolean[] visited = new boolean[26]; // Result array to build the subsequence char[] result = new char[text.length()]; // Stack pointer initialization int stackTop = -1;
- // from the stack marking it as not visited while (stackTop >= 0 && c < result[stackTop] && charCount[result[stackTop] - 'a'] > 0) { visited[result[stackTop] - 'a'] = false; stackTop--; // Push the current character onto the stack and mark it as visited

C++

public:

class Solution {

int presenceMask = 0; // Bitmask to keep track of characters included in the result. for (int i = 0; i < strSize; ++i) {</pre> char currentChar = s[i]; // The current character we're considering to add to the result. // Check if the character is already included in the result using the presence mask. if ((presenceMask >> (currentChar - 'a')) & 1) { continue; // If it's already present, move to the next character.

// While there are characters in the result, the last character in the result is // greater than the current character, and the last occurrence of the last character // in the result is after the current character in the original string... while (!result.empty() && result.back() > currentChar && lastIndexOf[result.back() - 'a'] > i) { // Remove the last character from the result since we've found // a better character to maintain lexicographical order. presenceMask ^= 1 << (result.back() - 'a'); // Update the presence mask.</pre> result.pop\_back(); // Remove the last character from the result.

// Add the current character to the result. // and update the presence mask accordingly. result.push back(currentChar); presenceMask |= 1 << (currentChar - 'a');</pre> // Once the loop is done, the 'result' contains the smallest lexicographical subsequence. return result; **}**; **TypeScript** function smallestSubsequence(s: string): string { // Function to get the index of a character in the alphabet

class Solution: def smallestSubsequence(self, s: str) -> str: # Dictionary to store the last occurrence index for each character last\_occurrence = {character: index for index, character in enumerate(s)} # Result stack to build the smallest subsequence stack = []# Set to keep track of characters already in the stack

while (stack.length > 0 && stack[stack.length - 1] > char && lastOccurrence[getIndex(stack[stack.length - 1])] > index) {

## **Time Complexity** The provided code's time complexity can be analyzed based on the number of iterations and operations that are performed:

the string once. Iterating through string s: The main for loop runs for every character, so it is O(n). However, we must also consider the

Building the last dictionary: This takes O(n) time, where n is the length of string s, as we go through every character of

- nested while loop. The nested while loop: Although there is a while loop inside the for loop, each element is added to the stk only once
- because of the vis set check, and each element is removed from stk only once. This means, in total, the while loop will run at most n times for the entire for loop. This does not change the overall time complexity which remains 0(n).

**Space Complexity** 

Analyzing the space used by the code: The last dictionary: The dictionary holds a mapping for each unique character to its last occurrence in s. In the worst case,

Combining these steps, the overall time complexity is O(n) where n is the length of the input string s.

- where all characters are unique, it will hold n entries, which leads to O(n) space. The stk list: As with the dictionary, in the worst case, it may hold all characters if they are all unique, leading to 0(n) space.
- The vis set: This also, in the worst-case scenario, will hold n entries for n unique characters in s, using 0(n) space. Considering all the auxiliary space used, the overall space complexity is O(n).