763. Partition Labels

Hash Table

Two Pointers

String

Problem Description

<u>Greedy</u>

Medium

The problem presents us with a string s. Our goal is to partition s into the maximum number of parts such that no letter appears in more than one part. This means that once a letter appears in a part, it cannot appear in any other part. After partitioning, when we concatenate all parts back together in the original order, the string should remain unchanged; it should be s.

Intuition

The resulting output should be a list of integers, where each integer represents the length of one of these partitions.

The intuition behind this solution is to first understand that a partition can only end when all instances of a particular character included in the partition have been encountered. The last occurrence of each character defines the maximum possible extent of a partition. To solve the problem, we track the last occurrence of each character using a dictionary. The keys are the characters from the

string s and the values are the indices of their last occurrences. As we iterate over the string s:

We continually update a mx variable to keep track of the farthest index that we must reach before we can end a partition. This

- is updated by looking at the last occurrence of the current character. When the current index i matches mx, it means we've included all occurrences of the characters seen so far in the current
- We append the size of the current partition to the answer list. The size is calculated as the difference between the current index i and the start index of the partition j, plus one.
- We then set the start of the next partition to be one index after the current index i.
- In the end, we get a list of partition sizes that fulfill the requirements.
- The solution uses a two-pass approach and utilizes a dictionary data structure to hold the last occurrence of each character in

partition. At this point, we can end the current partition and start a new one.

Building a dictionary of last occurrences: The first pass is over the string s. As we iterate, we capture the last position or

the string s.

Solution Approach

enumerate(s)}, making use of a dictionary comprehension. Partitioning based on last occurrences: The next part of the solution is where the actual partitioning logic is implemented:

index of every unique character in the dictionary called last. This is constructed in the line last = {c: i for i, c in

- We initialize three variables: mx, j, and ans. The mx variable represents the furthest point a partition can reach before being cut. The j variable is the start index of the current partition, and ans is the list that will hold the sizes of the partitions. We then iterate through the string s again, index by index, alongside the corresponding characters.
- For each character at index i, we update mx to be the maximum value between the current mx and the last occurrence of the character c, like this: mx = max(mx, last[c]).
- We calculate the size of the current partition by subtracting the start index j of this partition from the current index i and adding 1 (i - j + 1). This value is appended to the ans list. Once we add the partition size to ans, we set the start j of the next partition to i + 1.

■ If the current index i equals mx, it means all characters in the current partition will not appear again in the following parts of the string.

- Output the result: After the iteration completes, we return the list ans which contains the sizes of all partitions. The factors that make this algorithm work are:
- The dictionary captures the essential boundaries for our partitions by marking the last appearance of each character within the string. • The two-pass mechanism ensures that we never 'cut' a partition too early, which ensures that each letter only appears in one part.

• The greedy approach - always updating mx to extend the partition as far as possible and only finalizing a partition when absolutely necessary -

By following through with this approach, the function partitionLabels partitions the string optimally as per the problem statement.

Example Walkthrough

2.

ensures that we maximize the number of partitions.

Partitioning based on last occurrences:

○ We initialize mx to -1, j to 0, and ans to an empty list.

This is the right place to 'cut' this partition.

of every character. For our example, last would be {'a': 2, 'b': 1, 'c': 3}.

the partitions and to maximize the number of partitions. The output should be a list of the lengths of these partitions.

Let's consider a small example string s = "abac". We want to create partitions in such a way that no letters are repeated in any of

Building a dictionary of last occurrences: The first pass over the string s will give us a dictionary last with the last position

 We then loop through the string s, with the indices and characters. 1. At index 0, the character is $\frac{1}{a}$. We update $\frac{1}{a}$ to be the maximum of $\frac{1}{a}$ and the last occurrence of $\frac{1}{a}$ ($\frac{1}{a}$ = $\frac{1}{a}$), so $\frac{1}{a}$ becomes 2.

2. At index 1, the character is 'b'. We update mx to be max(2, 1) as the last occurrence of 'b' is at index 1. Since 2 is greater, mx remains

- 3. At index 2, the character is 'a'. Since we have already accounted for the last occurrence of 'a', we don't need to update mx it is still the
- right end of our partition. 4. At index 2 when i is equal to mx, we can 'cut' the partition here.

c. Hence, the final output is [3, 1].

- We append the size of this partition to ans, which is i j + 1 or 2 0 + 1, so we append 3. ■ We also set j to the next index i + 1, which is 3, ready for the next potential partition. 5. For the last character at index 3, 'c', the last occurrence is at 3 itself. We update mx to 3.
- 6. Given that i equals mx at the last character, we have the end of another partition. ■ The size of the partition is again calculated as i - j + 1, yielding 3 - 3 + 1, so we append 1 to ans.

Output the result: After the iteration completes, ans contains the sizes [3, 1], denoting the lengths of the partitions aba and

ensuring no letter appears in more than one part. The lengths of the partitions are [3, 1] as per the expected output. Solution Implementation

By following the steps above, the input string abac has been partitioned into ["aba", "c"], maximizing the number of parts and

class Solution: def partitionLabels(self, S: str) -> List[int]: # Create a dictionary to store the last occurrence of each character. last_occurrence = {char: index for index, char in enumerate(S)}

`max_last` represents the farthest index any character in the current partition has been seen.

Update `max_last` to be the max of itself and the last occurrence of the current character.

If the current index is the last occurrence of all characters seen so far in this partition,

`partition_start` represents the start of the current partition.

max_last = max(max_last, last_occurrence[char])

that means we can end the partition here.

Iterate through the characters of the string along with their indices.

This list will hold the sizes of the partitions. partition_sizes = []

Initialize variables.

max_last = partition_start = 0

for index, char in enumerate(S):

if max_last == index:

if (maxIndexSoFar == i) {

return partitionLengths;

partitionStart = i + 1;

// Add the size of the partition to the list.

partitionLengths.add(i - partitionStart + 1);

// Return the list containing the sizes of the partitions.

// Update the start index for the next partition.

Python

```
# Append the size of the current partition to the list (`index - partition_start + 1`).
                partition_sizes.append(index - partition_start + 1)
                # Update `partition_start` to the next index to start a new partition.
                partition_start = index + 1
       # Return the list of partition sizes.
        return partition_sizes
Java
import java.util.List;
import java.util.ArrayList;
class Solution {
    public List<Integer> partitionLabels(String s) {
       // Initialize an array to store the last appearance index of each character.
       int[] lastIndices = new int[26]; // Assuming 'a' to 'z' only appear in the string.
        int lengthOfString = s.length();
       // Fill the array with the index of the last occurrence of each character.
        for (int i = 0; i < lengthOfString; ++i) {</pre>
            lastIndices[s.charAt(i) - 'a'] = i;
       // Create a list to store the lengths of the partitions.
       List<Integer> partitionLengths = new ArrayList<>();
        int maxIndexSoFar = 0; // To keep track of the farthest reach of the characters seen so far.
        int partitionStart = 0; // The index where the current partition starts.
       // Iterate through the characters in the string.
        for (int i = 0; i < lengthOfString; ++i) {</pre>
           // Update the maxIndexSoFar with the farthest last occurrence of the current character.
           maxIndexSoFar = Math.max(maxIndexSoFar, lastIndices[s.charAt(i) - 'a']);
           // If the current index matches the maxIndexSoFar, we've reached the end of a partition.
```

C++

#include <vector>

```
#include <string>
using namespace std;
class Solution {
public:
    // This function partitions the string such that each letter appears in at
    // most one part, and returns a vector of integers representing the size of these parts.
    vector<int> partitionLabels(string S) {
        int lastIndex[26] = {0}; // Array to store the last index of each letter
        int length = S.size(); // Get the size of the string
       // Fill the array with the last index of each letter in the string
        for (int i = 0; i < length; ++i) {</pre>
            lastIndex[S[i] - 'a'] = i;
        vector<int> partitions; // This will store the sizes of the partitions
        int maxIndex = 0; // Maximum index of a character within the current partition
        int anchor = 0; // Starting index of the current partition
        for (int i = 0; i < length; ++i) {</pre>
            maxIndex = max(maxIndex, lastIndex[S[i] - 'a']); // Update maxIndex for the current partition
           // If the current index is the max index of the partition, it means we can make a cut here
            if (maxIndex == i) {
                partitions.push_back(i - anchor + 1); // Partition size is added to the result
                anchor = i + 1; // Update the starting index of the next partition
        return partitions; // Return the sizes of the partitions
};
TypeScript
function partitionLabels(inputString: string): number[] {
   // Initialize an array to hold the last index at which each letter appears.
    const lastIndex: number[] = Array(26).fill(0);
    // Helper function to convert a character to its relative position in the alphabet.
    const getCharacterIndex = (char: string) => char.charCodeAt(0) - 'a'.charCodeAt(0);
    // Get the length of the input string.
    const inputLength = inputString.length;
    // Update the lastIndex array with the last position for each letter in the input.
   for (let i = 0; i < inputLength; ++i) {</pre>
        lastIndex[getCharacterIndex(inputString[i])] = i;
    // Prepare an array to hold the lengths of the partitions.
```

```
// Return the array of partition sizes.
return partitionLengths;
```

const partitionLengths: number[] = [];

let start = 0, end = 0, maxIndex = 0;

if (maxIndex === i) {

start = i + 1;

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

// Initialize pointers for tracking the current partition.

partitionLengths.push(i - start + 1);

// Update the maximum index for the current partition.

// Add the size of the current partition to the array.

maxIndex = Math.max(maxIndex, lastIndex[getCharacterIndex(inputString[i])]);

// Move the start to the next position after the current partition.

// If the current position matches the maxIndex, a partition is complete.

```
class Solution:
   def partitionLabels(self, S: str) -> List[int]:
       # Create a dictionary to store the last occurrence of each character.
        last_occurrence = {char: index for index, char in enumerate(S)}
       # Initialize variables.
       # `max_last` represents the farthest index any character in the current partition has been seen.
       # `partition_start` represents the start of the current partition.
       max_last = partition_start = 0
       # This list will hold the sizes of the partitions.
       partition_sizes = []
       # Iterate through the characters of the string along with their indices.
       for index, char in enumerate(S):
           # Update `max_last` to be the max of itself and the last occurrence of the current character.
           max_last = max(max_last, last_occurrence[char])
           # If the current index is the last occurrence of all characters seen so far in this partition,
           # that means we can end the partition here.
           if max_last == index:
               # Append the size of the current partition to the list (`index - partition_start + 1`).
               partition sizes.append(index - partition start + 1)
               # Update `partition_start` to the next index to start a new partition.
               partition_start = index + 1
       # Return the list of partition sizes.
       return partition_sizes
Time and Space Complexity
Time Complexity:
  The time complexity for the provided code can be analyzed as follows:
```

runs in O(n) time, where n is the length of string s, because it iterates through the entire string once. • The subsequent for loop also runs through the string once and performs a constant number of operations within the loop. Hence, the

n.

partition on its own.

complexity of this part is O(n).

• Overall, since both the creation of the dictionary and the for loop run in linear time relative to the size of the input s, and occur sequentially, the total time complexity of the function is O(n).

• The first line within the partitionLabels function uses a dictionary comprehension to store the last appearance index of each character. This

- **Space Complexity:** The space complexity is influenced by the additional storage used:
- The last dictionary which stores the index of the last occurrence of each character has at most 0(min(n, 26)) space complexity because there are at most 26 letters in the alphabet. In the case where n < 26, the dictionary would have at most n elements.

• The ans list where the sizes of the partitions are stored will have a space complexity of O(n) in the worst case, where each character is a

• Since the last dictionary has a fixed upper limit due to a finite alphabet, it's typically considered 0(1) space, particularly in the context of large

• However, the list ans depends on the size of the input and can take up to 0(n) space in the worst case scenario. • Therefore, the overall space complexity of the function is O(n), reflecting the worst-case scenario where the output list could grow linearly with the input size.