2182. Construct String With Repeat Limit Medium Counting]

before we must use a different character to avoid breaking the repeat rule.

lexicographically largest possible string under the given constraints.

Heap (Priority Queue)

Leetcode Link

Problem Description

Greedy

String

the characters from s. The catch is that no character can repeat more than repeatLimit times consecutively. It is not necessary to use all characters from s. The goal is to build the lexicographically largest repeatLimitedString possible. A string is considered lexicographically larger if at the position where two strings start to differ, the string with the character higher

You are provided with a string s and an integer repeatLimit. The task is to create a new string, called repeatLimitedString, using

in the alphabet comes first. If they are identical up to the length of the shorter one, then the longer string is considered larger. Intuition

To form the lexicographically largest string, we should always prefer to use the largest character available (closest to 'z'). We start from the end of the alphabet and move towards the start ('z' to 'a'), using up to repeatLimit occurrences of the current character

character.

The intuition behind the solution is as follows: • Count the occurrences of each character in the input string s. This is done to quickly find out how many times we can use each

• Start constructing the repeatLimitedString by iterating from the highest character ('z') down to the lowest ('a'). Add up to

character without counting repeatedly.

- repeatLimit instances of the current character to the result. • If there are still occurrences of the current character left (more than repeatLimit), add a character next in line (the highest character remaining that is not the current one). This step ensures that we do not break the repeat limit for the current
- Repeat the process of adding the current character (up to the repeatLimit) and then the next in line (just once) until we run out of characters or we can no longer add the current character without breaking the repeat limit.
- If we reach a point where no other characters are available to be inserted between repeats of the current character, we terminate the process as we can't add more of the current character than allowed. By following this process and prioritizing the largest character possible at each step, we ensure that the result is the
- **Solution Approach**
- The solution approach involves a few key steps and makes use of a frequency counter and a greedy strategy to construct the required repeatLimitedString.

occurrences of each character in string s. The ASCII value is used to map characters 'a' to 'z' to indices 0 to 25.

2. Iterating in Reverse: We loop through the cnt array in reverse order starting from the index corresponding to 'z' (25) down to 'a' (0). This allows us to consider characters in decreasing lexicographic order.

3. Adding Characters to the Result: In each iteration, we add up to repeatLimit instances of the character corresponding to the

current index to the answer list ans, decrementing the count in cnt. The character to be added is obtained by computing

1. Counting Frequencies: We use an array cnt of size 26 (since there are 26 letters in the English alphabet) to count the

chr(ord('a') + i), where i is the current index.

Here's a portion of the code that visualizes this process:

ans.append(chr(ord('a') + j))

for _ in range(min(repeatLimit, cnt[i])):

for i in range(25, -1, -1):

if i < 0:

15 return ''.join(ans)

Example Walkthrough

1. Counting Frequencies:

constraints.

cnt[j] -= 1

12

13

Here's a step-by-step walkthrough of the implementation:

4. Maintaining the Repeat Limit: If the count of the current character is not exhausted after adding repeatLimit instances, we

count. We then add one instance of this character to ans, ensuring the repeat limit condition is satisfied.

character reaches 0 (cnt[i] == 0), or when there is no other character left to insert (j < 0).

- need to add a different character to ensure no character is repeated more than repeatLimit times in a row. 5. Finding the Next Character: We use another pointer j and decrement it to find the next available character with a nonzero
- By combining the usage of the frequency counter with a greedy approach, we ensure that we always construct the string with the highest lexicographical order possible under the given constraints of the repeatLimit.

6. Stopping Conditions: There are two stopping conditions in the while loop. The loop ends when the count of the current

ans.append(chr(ord('a') + i)) if cnt[i] == 0: while j >= 0 and cnt[j] == 0:

In the above code, and is a list that eventually contains the characters of the repeatLimitedString in the desired order. The join method is used to combine these characters into a string before returning it as the final answer.

Let's consider a small example to illustrate the solution approach. Suppose we have the string s = "aabab" and a repeatLimit of 2.

• We count each character's frequency and store it in an array cnt. Here's what the cnt array will look like for each alphabet

We want to construct the lexicographically largest repeatLimitedString by using the characters from s according to the given

```
• 'b': 2

    The other characters are not present in s and would have a count of 0.

2. Iterating in Reverse:
```

3. Adding Characters to the Result:

4. Maintaining the Repeat Limit:

6. Stopping Conditions:

given constraints.

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

41

42

43

Python Solution

class Solution:

consecutive 'a's.

Thus, we cannot add the last 'a'.

 $char_count = [0] * 26$

for i in range(25, -1, -1):

while True:

 $next_char_index = i - 1$

def repeatLimitedString(self, s: str, repeat_limit: int) -> str:

Iterate over characters from 'z' to 'a'

char_count[i] -= 1

next_char_index -= 1

public String repeatLimitedString(String s, int repeatLimit) {

// Array to hold the count of each letter in the string

// Counting the occurrences of each character in the string

if char_count[i] == 0:

if next_char_index < 0:</pre>

break

break

Return the constructed string

int[] letterCount = new int[26];

return ''.join(answer)

Keep placing characters until we run out

answer.append(chr(ord('a') + i))

Initialize a list to keep track of the count of each character

Find the next character to place if we hit the repeat limit

for _ in range(min(repeat_limit, char_count[i])):

Place the current character (up to the repeat limit) in the answer

Find the next highest character which we haven't exhausted

If there are no more characters to use, we are done

while next_char_index >= 0 and char_count[next_char_index] == 0:

If we have placed all instances of the current character, move to the next

• We decrement the count of 'b' in cnt to 0.

character:

• 'a': 3

5. Finding the Next Character: We have one 'a' remaining, but we cannot add it immediately because that would violate the repeatLimit. We search for the next available character, but as we've used up all 'b's, and no characters are left, we're only left with 'a'.

• We start from 'z' and go down to 'a', so the first character we consider from s that has a non-zero frequency is 'b'.

• We add 'b' to our result. Since repeatLimit is 2 and we have 2 'b's, we add both of them: repeatLimitedString = "bb".

Now that we've used all 'b's, we move to the next available character with the highest lexicographical value, which is 'a'.

• We can add up to 2 'a's (due to repeatLimit) to repeatLimitedString which now becomes: repeatLimitedString = "bbaa".

The lexicographically largest repeatLimitedString we can create with the given s and repeatLimit is bbaa.

By following the described solution approach, we constructed the lexicographically largest string without breaking the rules of the

We've reached the stopping condition because we have no characters left that we can insert to avoid having more than 2

5 6 # Count the occurrences of each character in the input string s for c in s: char_count[ord(c) - ord('a')] += 1 8 9 10 # Initialize a list to build the answer string 11 answer = []

```
36
37
                    # Place the next character
38
                    char_count[next_char_index] -= 1
                    answer.append(chr(ord('a') + next_char_index))
39
40
```

Java Solution

class Solution {

```
for (char ch : s.toCharArray()) {
                letterCount[ch - 'a']++;
10
11
12
           // StringBuilder to build the result string
           StringBuilder result = new StringBuilder();
13
14
15
           // Iterate from the letter 'z' to 'a'
16
            for (int i = 25; i >= 0; --i) {
               // Pointer to check for the next available smaller character
17
                int nextCharIndex = i - 1;
18
               while (true) {
20
                    // Append the current character up to repeatLimit times
21
22
                    for (int k = Math.min(repeatLimit, letterCount[i]); k > 0; --k) {
23
                        letterCount[i]--; // Decrement the count of the current character
24
                        result.append((char) ('a' + i)); // Add the current character to the result
25
26
27
                    // If all occurrences of the current character have been used up, break out of the loop
                    if (letterCount[i] == 0) {
                       break;
30
31
32
                    // Find the next available character which has at least one occurrence left
33
                    while (nextCharIndex >= 0 && letterCount[nextCharIndex] == 0) {
34
                        --nextCharIndex;
35
36
                    // If there are no more characters left to use, break out of the loop
37
                    if (nextCharIndex < 0) {</pre>
39
                        break;
40
                    // Append the next available character
                    letterCount[nextCharIndex]--;
43
                    result.append((char) ('a' + nextCharIndex));
45
46
           // Return the result as a string
           return result.toString();
49
50
51 }
52
C++ Solution
```

42 43 // Return the constructed string return result; 44 45 46 };

class Solution {

string repeatLimitedString(string s, int repeatLimit) {

vector<int> charCounts(26, 0);

charCounts[ch - 'a']++;

// Initialize the answer string

for (int i = 25; i >= 0; ---i) {

while (true) {

int nextCharIndex = i - 1;

charCounts[i]--;

--nextCharIndex;

for (char ch : s) {

string result;

// Count occurrences of each character in the input string

// Iterate over the characters starting from 'z' to 'a'

for (int k = repeatCount; k > 0; --k) {

result.push_back('a' + i);

if (charCounts[i] == 0) break;

if (nextCharIndex < 0) break;</pre>

charCounts[nextCharIndex]--;

result.push_back('a' + nextCharIndex);

// Continue to construct the string until all chars are used

// Add the current character up to repeatLimit times

int repeatCount = min(charCounts[i], repeatLimit);

// Break the loop if this character is fully used

// Break if there are no more characters to use

// Find the next available character with remaining count

// Add the next available character to the result string

while (nextCharIndex >= 0 && charCounts[nextCharIndex] == 0) {

public:

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

```
47
Typescript Solution
     function repeatLimitedString(s: string, repeatLimit: number): string {
         // Count occurrences of each character in the input string
         const charCounts: number[] = new Array(26).fill(0);
         for (const ch of s) {
             charCounts[ch.charCodeAt(0) - 'a'.charCodeAt(0)]++;
  6
  8
         // Initialize the answer string
         let result: string = '';
  9
 10
 11
         // Iterate over the characters starting from 'z' to 'a'
 12
         for (let i = 25; i >= 0; --i) {
 13
             let nextCharIndex = i - 1;
 14
 15
             // Continue to construct the string until all chars are used
 16
             while (true) {
 17
                 // Add the current character up to repeatLimit times
                 const repeatCount = Math.min(charCounts[i], repeatLimit);
 18
                 for (let k = repeatCount; k > 0; --k) {
 19
 20
                     charCounts[i]--;
 21
                     result += String.fromCharCode('a'.charCodeAt(0) + i);
 22
 23
 24
                 // Break the loop if this character is fully used
                 if (charCounts[i] === 0) break;
 25
 26
 27
                 // Find the next available character with remaining count
                 while (nextCharIndex >= 0 && charCounts[nextCharIndex] === 0) {
 28
                     --nextCharIndex;
 29
 30
 31
 32
                 // Break if there are no more characters to use
 33
                 if (nextCharIndex < 0) break;</pre>
 34
 35
                 // Add the next available character to the result string
 36
                 charCounts[nextCharIndex]--;
 37
                 result += String.fromCharCode('a'.charCodeAt(0) + nextCharIndex);
 38
 39
 40
 41
         // Return the constructed string
```

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

Time and Space Complexity

character after hitting the repeatLimit. However, since we only insert a single character in between repetitions and then

size.

continue with the same character until the repeatLimit is hit again, the actual number of visits to each character count (cnt[i])

Therefore, this part has a time complexity of O(n) where n is the length of the string s.

will be proportional to its frequency. This means that the operation count is effectively linear with respect to the string length n. Therefore, we can consider this part also to have a time complexity of O(n).

1. Character Counting: The first loop counts the frequency of each character in the string s. This loop runs exactly len(s) times.

2. Main Loop for Building the Answer: The outer loop runs 26 times (once for each lowercase English letter). The inner loop could,

in the worst case, run roughly n / repeatLimit times if all characters are the same and need to be interspersed with another

- (j) if the repeat limit is reached. In the worst-case scenario, this could traverse all characters smaller than i, adding an extra factor of at most 26 (constant time) per repeat limit hit. Thus, this does not affect the overall linear complexity relative to n. Hence the total time complexity is O(n) since the character search is bounded by a constant and doesn't depend on n. **Space Complexity**
- 2. Output List: The ans list ultimately stores each character from the input string in some order, and hence it will have a space

complexity of O(n). Thus, the total space complexity of the algorithm is O(n) due to the output list size being proportional to the input size.

- return result; 42 43 44
 - 3. Looking for the Next Character: Within the inner while loop, there's another loop that searches for the next available character
 - The space complexity is determined by the additional space used by the algorithm which is not part of the input or output: 1. Character Count Array: The cnt array has a space complexity of 0(1) since it is always a fixed size of 26 regardless of the input