767. Reorganize String

281 19

Description (/problems/reorganize-string/description/) Hints (/problems/reorganize-string/hints/) Submissions (/problems/reorganize-string/s

Quick Navigation View in Article (/articles/reorganized-string/) Notes

Approach #1: Sort by Count [Accepted]

Intuition

If we should make no two `'a'` s adjacent, it is natural to write `"aXaXaXa..."` where `"X"` is some letter. For now, let's assume that the task is possible (ie. the answer is not `""` .)

Let's sort the string `S` , so all of the same kind of letter occur in continuous *blocks*. Then when writing in the following interleaving pattern, like `S[3], S[0], S[4], S[1], S[5], S[2]` , adjacent letters never touch. (The specific interleaving pattern is that we start writing at index 1 and step by 2; then start from index 0 and step by 2.)

The exception to this rule is if `N` is odd, and then when interleaving like `S[2], S[0], S[3], S[1], S[4]` , we might fail incorrectly if there is a block of the same 3 letters starting at `S[0]` or `S[1]` . To prevent failing erroneously in this case, we need to make sure that the most common letters all occur at the end.

Finally, it is easy to see that if `N` is the length of the string, and the count of some letter is greater than `(N+1) / 2` , the task is impossible.

Algorithm

Find the count of each character, and use it to sort the string by count.

If at some point the number of occurrences of some character is greater than `(N + 1) / 2` , the task is impossible.

Otherwise, interleave the characters in the order described above.

Java Python Copy

```java
class Solution {
    public String reorganizeString(String S) {
        int N = S.length();
        int[] counts = new int[26];
        for (char c: S.toCharArray()) counts[c-'a'] += 100;
        for (int i = 0; i < 26; ++i) counts[i] += i;
        //Encoded counts[i] = 100*(actual count) + (i)
        Arrays.sort(counts);

        char[] ans = new char[N];
        int t = 1;
        for (int code: counts) {
            int ct = code / 100;
            char ch = (char) ('a' + (code % 100));
            if (ct > (N+1) / 2) return "";
            for (int i = 0; i < ct; ++i) {
                if (t >= N) t = 0;
                ans[t] = ch;
                t += 2;
            }
        }

        return String.valueOf(ans);
    }
}
```

Complexity Analysis

- Time Complexity: $O(\mathcal{A}(N + \log \mathcal{A}))$, where N is the length of S, and $\mathcal{A}$ is the size of the alphabet. In Java, our implementation is $O(N + \mathcal{A} \log \mathcal{A})$. If $\mathcal{A}$ is fixed, this complexity is $O(N)$.
- Space Complexity: $O(N)$. In Java, our implementation is $O(N + \mathcal{A})$.

Approach #2: Greedy with Heap [Accepted]

Intuition

One consequence of the reasoning in *Approach #1*, is that a greedy approach that tries to write the most common letter (that isn't the same as the previous letter written) will work.

The reason is that the task is only impossible if the frequency of a letter exceeds `(N+1) / 2`. Writing the most common letter followed by the second most common letter keeps this invariant.

A heap is a natural structure to repeatedly return the current top 2 letters with the largest remaining counts.

Approach

We store a heap of (count, letter). [In Python, our implementation stores negative counts.]

We pop the top two elements from the heap (representing different letters with positive remaining count), and then write the most frequent one that isn't the same as the most recent one written. After, we push the correct counts back onto the heap.

Actually, we don't even need to keep track of the most recent one written. If it is possible to organize the string, the letter written second can never be written first in the very next writing.

At the end, we might have one element still on the heap, which must have a count of one. If we do, we'll add that to the answer too.

Java Python Copy

```java
class Solution {
    public String reorganizeString(String S) {
        int N = S.length();
        int[] count = new int[26];
        for (char c: S.toCharArray()) count[c-'a']++;
        PriorityQueue<MultiChar> pq = new PriorityQueue<MultiChar>((a, b) ->
            a.count == b.count ? a.letter - b.letter : b.count - a.count);

        for (int i = 0; i < 26; ++i) if (count[i] > 0) {
            if (count[i] > (N + 1) / 2) return "";
            pq.add(new MultiChar(count[i], (char) ('a' + i)));
        }

        StringBuilder ans = new StringBuilder();
        while (pq.size() >= 2) {
            MultiChar mc1 = pq.poll();
            MultiChar mc2 = pq.poll();
            /*This code turns out to be superfluous, but explains what is happening
            if (ans.length() == 0 || mc1.letter != ans.charAt(ans.length() - 1)) {
                ans.append(mc1.letter);
                ans.append(mc2.letter);
            } else {
                ans.append(mc2.letter);
                ans.append(mc1.letter);
            }*/
            ans.append(mc1.letter);
            ans.append(mc2.letter);
```

Complexity Analysis

- Time Complexity: $O(N \log \mathcal{A}))$, where N is the length of S, and $\mathcal{A}$ is the size of the alphabet. If $\mathcal{A}$ is fixed, this complexity is $O(N)$.
- Space Complexity: $O(\mathcal{A})$. If $\mathcal{A}$ is fixed, this complexity is $O(1)$.

Analysis written by: @awice (https://leetcode.com/awice).