Strong Password Checker

A password is considered strong if below conditions are all met:

1. It has at least 6 characters and at most 20 characters.
2. It must contain at least one lowercase letter, at least one uppercase letter, and at least one digit.
3. It must NOT contain three repeating characters in a row ("...aaa..." is weak, but "...aa...a..." is strong, assuming other conditions are met).

Write a function strongPasswordChecker(s), that takes a string s as input, and return the **MINIMUM** change required to make s a strong password. If s is already strong, return 0.

Insertion, deletion or replace of any one character are all considered as one change.

Solution 1

I've separated the problem into three cases:
(1) `s.length() < 6`
(2) `6 <= s.length() <= 20`
(3) `s.length() > 20`

Let's look at case (1) first. If `s.length() < 6`, we know we have room to insert some more letters into `s`. Question is how to use the insertions effectively to reduce the number of potential replacements. I'm using a greedy approach for this one: I'm inserting one char between the second and third chars whenever I see a repetition of 3 letters as substring.

e.g. Say we have room to insert some chars in string and we see a substring of `"aaaa"`. I'll insert a `'B'` to make it `"aaBaa"` to break the 3-char repetition, thus reducing potential replacement by 1. And we'll do this until we can't insert any more chars into `s`. When we reach this point, we'll start dealing with case (2)

For case (2), I still follow a greedy approach. I'm simply searching for 3-char repetitions, and replacing one of the chars to break the repetition.
e.g. If we see a substring of `"aaaa"`, we'll make it `"aaBa"`.

My code deals with (1) and (2) together as `s.length() <= 20`.

Case (3) is a little bit tricky because simple greedy doesn't work any more.
When `s.length() > 20`, we want to delete some chars instead of inserting chars to reduce potential replacements. Question is the same: how to do it effectively? Let's do some observations here:

Say `len` is the length of each repetition.
(a) `len % 3` only has three possible values, namely 0, 1 and 2.
(b) Minimum number of replacements needed to break each repetition is `len / 3`.
(c) Based on (a) and (b), we know that deletion can reduce replacements only if the deletion can change the value of `len / 3`
(d) Based on (c), we know if we want to reduce 1 replacement, we need 1 deletion for `len % 3 == 0`, and 2 deletions for `len % 3 == 1`, and 3 deletions for `len % 3 == 2`.

Given above observations, I simply implemented the solution to do (d).

Also note that missing of upper case char, lower case char, or digit can always be resolved by insertion or replacement.

I've pasted two versions of the solutions below, with and without comments, for easier reference.

Without comments:

```
class Solution {
public:
    int strongPasswordChecker(string s) {
        int deleteTarget = max(0, (int)s.length() - 20), addTarget = max(0, 6 - (
int)s.length());
        int toDelete = 0, toAdd = 0, toReplace = 0, needUpper = 1, needLower = 1,
needDigit = 1;

        for (int l = 0, r = 0; r < s.length(); r++) {
            if (isupper(s[r])) { needUpper = 0; }
            if (islower(s[r])) { needLower = 0; }
            if (isdigit(s[r])) { needDigit = 0; }

            if (r - l == 2) {
                if (s[l] == s[l + 1] && s[l + 1] == s[r]) {
                    if (toAdd < addTarget) { toAdd++, l = r; }
                    else { toReplace++, l = r + 1; }
                } else { l++; }
            }
        }
        if (s.length() <= 20) { return max(addTarget + toReplace, needUpper + nee
dLower + needDigit); }

        toReplace = 0;
        vector<unordered_map<int, int>> lenCnts(3);
        for (int l = 0, r = 0, len; r <= s.length(); r++) {
            if (r == s.length() || s[l] != s[r]) {
                if ((len = r - l) > 2) { lenCnts[len % 3][len]++; }
                l = r;
            }
        }

        for (int i = 0, numLetters, dec; i < 3; i++) {
            for (auto it = lenCnts[i].begin(); it != lenCnts[i].end(); it++) {
                if (i < 2) {
                    numLetters = i + 1, dec = min(it->second, (deleteTarget - toD
elete) / numLetters);
                    toDelete += dec * numLetters, it->second -= dec;

                    if (it->first - numLetters > 2) { lenCnts[2][it->first - numL
etters] += dec; }
                }
                toReplace += (it->second) * ((it->first) / 3);
            }
        }

        int dec = (deleteTarget - toDelete) / 3;
        toReplace -= dec, toDelete -= dec * 3;
        return deleteTarget + max(toReplace, needUpper + needLower + needDigit);
    }
};
```

With comments:

```
class Solution {
public:
    int strongPasswordChecker(string s) {
        int deleteTarget = max(0, (int)s.length() - 20), addTarget = max(0, 6 - (
int)s.length());
        int toDelete = 0, toAdd = 0, toReplace = 0, needUpper = 1, needLower = 1,
needDigit = 1;

        ////////////////////////////////
        // For cases of s.length() <= 20 //
        ////////////////////////////////
        for (int l = 0, r = 0; r < s.length(); r++) {
            if (isupper(s[r])) { needUpper = 0; }
            if (islower(s[r])) { needLower = 0; }
            if (isdigit(s[r])) { needDigit = 0; }

            if (r - l == 2) {                                   // if it's a thr
ee-letter window
                if (s[l] == s[l + 1] && s[l + 1] == s[r]) {     // found a three-
repeating substr
                    if (toAdd < addTarget) { toAdd++, l = r; }  // insert letter
to break repetition if possible
                    else { toReplace++, l = r + 1; }            // replace curre
nt word to avoid three repeating chars
                } else { l++; }                                 // keep the wind
ow with no more than 3 letters
            }
        }
        if (s.length() <= 20) { return max(addTarget + toReplace, needUpper + nee
dLower + needDigit); }

        ////////////////////////////////
        // For cases of s.length() > 20 //
        ////////////////////////////////
        toReplace = 0;                                          // reset toRepla
ce
        vector<unordered_map<int, int>> lenCnts(3);             // to record repe
titions with (length % 3) == 0, 1 or 2
        for (int l = 0, r = 0, len; r <= s.length(); r++) {     // record all rep
etion frequencies
            if (r == s.length() || s[l] != s[r]) {
                if ((len = r - l) > 2) { lenCnts[len % 3][len]++; } // we only ca
re about repetions with length >= 3
                l = r;
            }
        }

        /*
            Use deletions to minimize replacements, following below orders:
            (1) Try to delete one letter from repetitions with (length % 3) == 0.
Each deletion decreases replacement by 1
            (2) Try to delete two letters from repetitions with (length % 3) == 1
. Each deletion decreases repalcement by 1
            (3) Try to delete multiple of three letters from repetions with (leng
th % 3) == 2. Each deletion (of three
            letters) decreases repalcements by 1
```

```
        */
        for (int i = 0, numLetters, dec; i < 3; i++) {
            for (auto it = lenCnts[i].begin(); it != lenCnts[i].end(); it++) {
                if (i < 2) {
                    numLetters = i + 1, dec = min(it->second, (deleteTarget - toD
elete) / numLetters);
                    toDelete += dec * numLetters;               // dec is the num
ber of repetitions we'll delete from
                    it->second -= dec;                          // update number
of repetitions left

                    // after letters deleted, it fits in the group where (length
% 3) == 2
                    if (it->first - numLetters > 2) { lenCnts[2][it->first - numL
etters] += dec; }
                }

                // record number of replacements needed
                // note if len is the length of repetition, we need (len / 3) num
ber of replacements
                toReplace += (it->second) * ((it->first) / 3);
            }
        }

        int dec = (deleteTarget - toDelete) / 3;                // try to delete
multiple of three letters as many as possible
        toReplace -= dec, toDelete -= dec * 3;
        return deleteTarget + max(toReplace, needUpper + needLower + needDigit);
    }
};
```

written by soamaaazing original link here

Solution 2

There are 2 wrong test cases which I list at the beginning.

The general idea is to record some states, and calculate the edit distance at the end. All detail are explained in the comments.

```java
public class Solution {
    public int strongPasswordChecker(String s) {

        if(s.length()<2) return 6-s.length();

        //Initialize the states, including current ending character(end), existen
ce of lowercase letter(lower), uppercase letter(upper), digit(digit) and number o
f replicates for ending character(end_rep)
        char end = s.charAt(0);
        boolean upper = end>='A'&&end<='Z', lower = end>='a'&&end<='z', digit = e
nd>='0'&&end<='9';

        //Also initialize the number of modification for repeated characters, tot
al number needed for eliminate all consequnce 3 same character by replacement(cha
nge), and potential maximun operation of deleting characters(delete). Note delete
[0] means maximum number of reduce 1 replacement operation by 1 deletion operatio
n, delete[1] means maximun number of reduce 1 replacement by 2 deletion operation
, delete[2] is no use here.
        int end_rep = 1, change = 0;
        int[] delete = new int[3];

        for(int i = 1;i<s.length();++i){
            if(s.charAt(i)==end) ++end_rep;
            else{
                change+=end_rep/3;
                if(end_rep/3>0) ++delete[end_rep%3];
                //updating the states
                end = s.charAt(i);
                upper = upper||end>='A'&&end<='Z';
                lower = lower||end>='a'&&end<='z';
                digit = digit||end>='0'&&end<='9';
                end_rep = 1;
            }
        }
        change+=end_rep/3;
        if(end_rep/3>0) ++delete[end_rep%3];

        //The number of replcement needed for missing of specific character(lower
/upper/digit)
        int check_req = (upper?0:1)+(lower?0:1)+(digit?0:1);

        if(s.length()>20){
            int del = s.length()-20;

            //Reduce the number of replacement operation by deletion
            if(del<=delete[0]) change-=del;
            else if(del-delete[0]<=2*delete[1]) change-=delete[0]+(del-delete[0])
/2;
            else change-=delete[0]+delete[1]+(del-delete[0]-2*delete[1])/3;

            return del+Math.max(check_req,change);
        }
        else return Math.max(6-s.length(), Math.max(check_req, change));
    }
}
```

The author is already fixed all the test cases.

written by danzhutest original link here

Solution 3

The basic principle is straightforward: if we want to make MINIMUM changes to turn `s` into a strong password, each change made should fix as many problems as possible.

So to start, let's first identify all the problems in the input string `s` and list what changes are suitable for righting each of them. To clarify, each change should be characterized by at least two parts: the **type of operation** it takes and the **position in the string** where the operation is applied (`Note`: Ideally we should also include the characters involved in the operation and the "power" of each operation for eliminating problems but they turn out to be partially relevant so I will mention them only when appropriate).

1. `Length problem`: if the total length is **less than 6**, the change that should be made is (**insert, any position**), which reads as "the operation is insertion and it can be applied to anywhere in the string". If the total length is **greater than 20**, then the change should be (**delete, any position**).
2. `Missing letter or digit`: if any of the lowercase/uppercase letters or digits is missing, we can do either (**insert, any position**) or (**replace, any position**) to correct it. (`Note` here the characters for insertion or replacement can only be those missing.)
3. `Repeating characters`: for repeating characters, all three operations are allowed but the positions where they can be applied are limited within the repeating characters. For example, to fix `"aaaaa"`, we can do one replacement (replace the middle `'a'`) or two insertions (one after the second `'a'` and one after the fourth `'a'`) or three deletions (delete any of the three `'a'`s). So the possible changes are (**replace, repeating characters**), (**insert, repeating characters**), (**delete, repeating characters**). (`Note` however the "power" of each operation for fixing the problem are different -- replacement is the strongest while deletion is the weakest.)

All right, what's next? If we want a change to eliminate as many problems as it can, it must be shared among the possible solutions to each problem it can fix. So our task is to find out possible overlapping among the changes for fixing each problem.

Since there are most (three) changes allowed for the third problem, we may start from combinations `first problem & third problem` and `second problem & third problem`. It's not too hard to conclude that any change that can fix the first or second problems is also able to fix the third one (since the type of operation here is irrelevant, we are free to choose the position of the operation to match those of the repeating characters). For combination `first problem & second problem`, depending on the length of the string, there will be overlapping if length is less than 6 or no overlapping if length is greater than 20.

From the analyses above, it seems worthwhile to distinguish between the two cases: when the input string is too short or too long.

For the former case, it can be shown that the total changes needed to fix the first and second problems always outnumber those for the third one. Since whatever change used fixing the first two problems can also correct the third one, we may concern ourselves with only the first two. Also as there is overlapping between the changes for fixing the first two problems, we will prefer those overlapping ones, i.e. (**insert, any position**). Another point is that the characters involved in the operation matters now. To fix the first problem, only those missing characters can be inserted while for the second condition, it can be any character. Therefore correcting the first problem takes precedence over the second one.

For the latter case, there is overlapping between the**first & third** and **second & third** problems, so those overlapping changes will be taken, i.e., first problem => (**delete, any position**), second problem => (**replace, any position**). The reason not to use (**insert, any position**) for the second problem is that it contradicts the changes made to the first problem (therefore has the tendency to cancel its effects). After fixing the first two problems, what operation(s) should we choose for the third one?

Now the "power" of each operation for eliminating problems comes into play. For the third problem, the "power" of each operation will be measured by the maximum number of repeating characters it is able to get rid of. For example, one replacement can eliminate at most `5` repeating characters while insertion and deletion can do at most `4` and `3`, respectively. In this case, we say replacement has more "power" than insertion or deletion. Intuitively the more "powerful" the operation is, the less number of changes is needed for correcting the problem. Therefore (**replace, repeating characters**) triumphs in terms of fixing the third problem.

Furthermore, another very interesting point shows up when the "power" of operation is taken into consideration (And thank **yicui** for pointing it out). As I mentioned that there is overlapping between changes made for fixing the first two problems and for the third one, which means the operations chosen above for the first two problems will also be applied to the third one. For the second problem with change chosen as (**replace, any position**), we have no problem adapting it so that it coincides with the optimal change (**replace, repeating characters**) made for the third problem. However, there is no way to do the same for the first problem with change (**delete, any position**). We have a conflict now!

How do we reconcile it? The trick is that for a sequence of repeating characters of length `k` (`k >= 3`), instead of turning it all the way into a sequence of length `2` (so as to fix the repeating character problem) by the change (**delete, any position**), we will first reduce its length to (`3m + 2`), where (`3m + 2`) is the largest integer of the form yet no more than `k`. That is to say, if `k` is a multiple of `3`, we apply once such change so its length will become (`k - 1`); else if `k` is a multiple of `3` plus `1`, we apply twice such change to cut its length down to (`k - 2`), provided we have more such changes to spare (be careful here as we need at least two changes but the remaining available changes may be less than that, so we should stick to the smaller one: `2` or the remaining available changes). The reason is that the optimal change (**replace, repeating characters**) for the third problem will be most "powerful"

when the total length of the repeating characters is of this form. Of course, if we still have more changes (**delete, any position**) to do after that, then we are free to turn the repeating sequence all the way into a sequence of length `2`.

Here is the java program based on the above analyses. Both time and space complexity is `O(n)`. Not sure if we can reduce the space down to `O(1)` by computing the `arr` array on the fly. A quick explanation is given at the end.

```java
public int strongPasswordChecker(String s) {
    int res = 0, a = 1, A = 1, d = 1;
    char[] carr = s.toCharArray();
    int[] arr = new int[carr.length];

    for (int i = 0; i < arr.length;) {
        if (Character.isLowerCase(carr[i])) a = 0;
        if (Character.isUpperCase(carr[i])) A = 0;
        if (Character.isDigit(carr[i])) d = 0;

        int j = i;
        while (i < carr.length && carr[i] == carr[j]) i++;
        arr[j] = i - j;
    }

    int total_missing = (a + A + d);

    if (arr.length < 6) {
        res += total_missing + Math.max(0, 6 - (arr.length + total_missing));

    } else {
        int over_len = Math.max(arr.length - 20, 0), left_over = 0;
        res += over_len;

        for (int k = 1; k < 3; k++) {
            for (int i = 0; i < arr.length && over_len > 0; i++) {
                if (arr[i] < 3 || arr[i] % 3 != (k - 1)) continue;
                arr[i] -= Math.min(over_len, k);
                over_len -= k;
            }
        }

        for (int i = 0; i < arr.length; i++) {
            if (arr[i] >= 3 && over_len > 0) {
                int need = arr[i] - 2;
                arr[i] -= over_len;
                over_len -= need;
            }

            if (arr[i] >= 3) left_over += arr[i] / 3;
        }

        res += Math.max(total_missing, left_over);
    }

    return res;
}
```

A quick explanation of the program:

1. `res` denotes the minimum changes; `a`, `A` and `d` are the number of missing lowercase letters, uppercase letters and digits, respectively; `arr` is an integer array whose element will be the number of repeating characters starting at the corresponding position in the string.

2. In the following loop we fill in the values for `a`, `A`, `d` and `arr` to identify the problems for each condition. The total number of missing characters `total_missing` will be the summation of `a`, `A`, `d` and fixing this problem takes at least `total_missing` changes.
3. We then distinguish the two cases when the string is too short or too long. If it is too short, we pad its length to at least `6` (note in this case we've already inserted `total_missing` characters so the new length is the summation of the original length and `total_missing`).
4. Otherwise, to fix the first condition, we need to delete `over_len` (number of surplus characters) characters. Since fixing the first problem also corrects the third one, we need to get rid of those parts from the `arr` array. And as I mentioned, we need to first turn all numbers in the `arr` array greater than `2` into the form of (`3m + 2`) and then reduce them all the way to `2` if `over_len` is still greater than `0`. After that, we need to replace `total_missing` characters to fix the second problem, which also fixes part (or all) of the third problem. Therefore we only need to take the larger number of changes needed for fixing the second problem (which is `total_missing`) and for the third one (which is `left_over`, since it is the number of changes still needed after fixing the first problem).

written by fun4LeetCode original link here