How to find the minimum number of operation(s) to make the string balanced?

7

4

From Codechef:

> A string is considered *balanced* if and only if all the characters occur in it equal number of times.
>
> You are given a string `s` ; this string may only contain uppercase English letters. You may perform the following operation any number of times (including zero): Choose one letter in `s` and replace it by another uppercase English letter. Note that even if the replaced letter occurs in `s` multiple times, only the chosen occurrence of this letter is replaced.
>
> Find the minimum number of operations required to convert the given string to a balanced string.
>
> Example:
>
> For input: `ABCB`

> So, the number of minimum operation(s)= `1` .

How to make the string good?

Can I apply dynamic programming to it ?

algorithm dynamic-programming

edited Feb 4 at 12:15

Bart Kiers
132k 28 244 248

asked Feb 2 at 9:48

A

Ananye Agarwal
51 7

locked by Yvette Colomb ♦ Feb 4 at 12:24

This post has been locked while disputes about its content are being resolved. For more info visit meta.

Read more about locked posts here.

Is it valid to change `B` to `D` , and get `ABCD` , I mean is it allowed to use a char not exists in original string to replace an existing char ? – Eric Wang Feb 2 at 12:14

@Anaye why do you keep removing most of your question? – Bart Kiers Feb 4 at 12:13

Please do not vandalise your post. This may result in a question ban. By posting on the Stack Exchange network, you've granted a non-revocable right for SE to distribute that content (under the CC BY-SA 3.0 license). By SE policy, any vandalism will be reverted. If you would like to disassociate this post from your account, see What is the proper route for a

2 Answers

11

I don't think you really need dynamic programming here.

One approach in *O*(length(*S*)) time:

- Iterate over *S*, build a frequency-map (a mapping from distinct letters A–Z to counts). For your `ABCB` example, that would be `A->1 B->2 C->1 D->0 E->0 ... Z->0`, which we can represent as the array `[1, 2, 1, 0, 0, ..., 0]`.
  - We can do this because we don't actually care about the positions of letters at all; there's no real difference between `ABCB` and `ABBC`, in that each can be balanced by replacing their `C` with an `A`.
- Sort the array. For your example, that gives `[0, 0, ..., 0, 1, 1, 2]`.
  - We can do this because we don't actually care about which letter was which; there's no real difference between `ABCB` and `ABDB`, in that each can be balanced by replacing one of their singleton letters with their other one.
- Assuming the string is nonempty (since if it is then the answer is just 0), the final balanced string will contain at least 1 and at most 26 distinct letters. For each integer *i* between 1 and 26, determine how many operations you'd need to perform in order to produce a balanced string with *i* distinct letters:
  - First, check that length(*S*) is a multiple of *i*; if not, this isn't possible, so skip to the next integer.
  - Next, compute $^{\text{length}(S)}/_{i}$, the count of each distinct letter in the final balanced string.
  - To count how many operations need to be performed, we're going to examine all the counts that need to be

- We're only interested in the last *i* elements of the sorted array. Any elements before that point represent letters that won't occur in the balanced string: either the counts are already zero and will remain so, or they are nonzero but will be decreased to zero. Either way, since we're only tracking *increases*, we can ignore them.
- For each of the last *i* counts that are less than $\text{length}(S)/i$, add the difference. This sum is the number of operations. (Note that, since the counts are sorted, you can exit this inner loop as soon as you get to a count that's greater than or equal to the target count.)
- You can exit this loop after the first *i* that's greater than or equal to the number of distinct letters in the original *S* (aside from values of *i* that we had to skip because they don't evenly divide length(*S*)). For example, if length(*S*) = 100, and the original *S* has 19 distinct letters, then we only need to consider *i* as high as 20. (Hat-tip to Eric Wang for suggesting something along these lines.)

- Return the least of these up-to-26 sums. (Note that you don't actually need to store all the sums; you can just keep track of the minimum.)

edited Feb 2 at 22:10

answered Feb 2 at 11:02

ruakh
125k 13 201 256

but this method works in all cases to find the best possible answer :-) Observations!! – Ananye Agarwal Feb 2 at 12:47

Can you give some ideas on this one :-

@AnanyeAgarwal: I actually don't have a great answer for that one. The DP part seems straightforward -- once you've computed the minimal number of steps needed to bring all of S[i..j] to the same letter, you can do *K* passes to find the minimal number of steps to bring f(S[1..j]) down to at most *K*. Since K·|S| ≤ 65,536, that second part should be quite fast. But I can't think of any way to do the first part in fewer than 26·|S|²/2 steps, which seems like it's probably too slow. – ruakh 2 days ago

That's really cool. And in case anyone missed it, this approach also covers Minimum number of adjustments to make array elements equal or 0 by equating the string length with the invariant sum, and the alphabet with the array's indexes. – גלעד ברקן 2 days ago

I was solving same problem, I thought upto using 1 to 26, but didn't get idea of sorting.Nice!! – PRY 2 days ago

1

Following code implement the solution, in Java, with unit test.

The algorithm is almost identical as in @ruakh's answer, if not identical.

Code

BalanceString.java

```
import java.util.Arrays;
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;
```

```
 * <p>figure out the minimum replacement required to make the string
 * <p>which means each char in the string occurs the same time,
 *
 * @author eric
 * @date 2/2/19 8:54 PM
 */
public class BalanceString {
    private final char minChar;
    private final char maxChar;
    private final int distinctChars; // total distinct char count,

    public static final BalanceString EN_UPPER_INSTANCE = new Balance

    public BalanceString(char minChar, char maxChar) {
        this.minChar = minChar;
        this.maxChar = maxChar;
        this.distinctChars = maxChar - minChar + 1;

        if (distinctChars <= 0)
            throw new IllegalArgumentException("invalid range of char
" + maxChar + "]");
    }

    /**
     * Check minimal moves needed to make string balanced.
     *
     * @param str
     * @return count of moves
     */
    public int balanceCount(String str) {
        // System.out.printf("string to balance:\t%s\n", str);
        int len = str.length(); // get length,
        if (len <= 2) return 0; // corner cases,

        Map<Character, Integer> coMap = figureOccurs(str); // figure
        Integer[] occurs = sortOccursReversely(coMap); // reversely o

        int m = coMap.size(); // distinct char count,
        int maxN = (len < distinctChars ? len : distinctChars); // ge
distinct char count, included,

        int smallestMoves = Integer.MAX_VALUE; // smallest moves, amo

        // check each possible n, and get its moves,
```

```
            if (moves < smallestMoves) smallestMoves = moves;
        }
    }

    return smallestMoves;
}

/**
 * Figure occurs for each char.
 *
 * @param str
 * @return
 */
protected Map<Character, Integer> figureOccurs(String str) {
    Map<Character, Integer> coMap = new HashMap<>();
    for (char c : str.toCharArray()) {
        if (c < minChar || c > maxChar)
            throw new IllegalArgumentException(c + " is not withi

        if (!coMap.containsKey(c)) coMap.put(c, 1);
        else coMap.put(c, coMap.get(c) + 1);
    }

    return coMap;
}

/**
 * Get reverse sorted occurrences.
 *
 * @param coMap
 * @return
 */
protected Integer[] sortOccursReversely(Map<Character, Integer> c
    Integer[] occurs = new Integer[coMap.size()];

    coMap.values().toArray(occurs);
    Arrays.sort(occurs, Collections.reverseOrder());

    return occurs;
}

/**
 * Figure moves needed to balance.
 *
```

```
     * @param n     new distinct elements count,
     * @return count of moves,
     */
    protected int figureMoves(int len, Map<Character, Integer> coMap,
int m, int n) {
        int avgOccur = len / n; // average occurrence,
        int moves = 0;

        if (n == m) { // distinct chars don't change,
            for (Integer occur : occurs) {
                if (occur <= avgOccur) break;
                moves += (occur - avgOccur);
            }
        } else if (n < m) { // distinct chars decrease,
            for (int i = 0; i < n; i++) moves += Math.abs(occurs[i] -
elements kept,
            for (int i = n; i < m; i++) moves += occurs[i]; // for el
            moves >>= 1;
        } else { // distinct chars increase,
            for (int i = 0; i < occurs.length; i++) moves += Math.abs
avgOccur); // for existing elements,
            moves += ((n - m) * avgOccur); // for new elements,
            moves >>= 1;
        }

        return moves;
    }

    public char getMinChar() {
        return minChar;
    }

    public char getMaxChar() {
        return maxChar;
    }

    public int getDistinctChars() {
        return distinctChars;
    }
}
```

BalanceStringTest.java
(Unit test, via `TestNG`)

```
/**
 * BalanceString test.
 *
 * @author eric
 * @date 2/2/19 9:36 PM
 */
public class BalanceStringTest {
    private BalanceString bs = BalanceString.EN_UPPER_INSTANCE;

    @Test
    public void test() {
        // n < m case,
        Assert.assertEquals(bs.balanceCount("AAAABBBC"), 1); // e.g 1
        Assert.assertEquals(bs.balanceCount("AAAAABBC"), 2); // e.g 1
        Assert.assertEquals(bs.balanceCount("AAAAAABC"), 2); // e.g 1
        Assert.assertEquals(bs.balanceCount("AAAAAAAB"), 1); // e.g 1

        // n > m case,
        Assert.assertEquals(bs.balanceCount("AAAABBBBCCCCDDDDEEEEAAAA
char, e.g change 4 A to 4 F,
        Assert.assertEquals(bs.balanceCount(genIncSeq(10)), 15); // A
chars, 55 in length; solved by add 1 new char, need 15 steps,

        // n == m case,
        Assert.assertEquals(bs.balanceCount(genIncSeq(3)), 1); // A-C
6 in length; symmetric, solved with same distinct chars, need 1 steps
        Assert.assertEquals(bs.balanceCount(genIncSeq(11)), 15); // A
chars, 66 in length; symmetric, solved with same distinct chars, need

        // n < m, or n > m case,
        Assert.assertEquals(bs.balanceCount("ABAC"), 1); // e.g 1A ->
    }

    // corner case,
    @Test
    public void testCorner() {
        // m <= 2,
        Assert.assertEquals(bs.balanceCount(""), 0);
        Assert.assertEquals(bs.balanceCount("A"), 0);
        Assert.assertEquals(bs.balanceCount("AB"), 0);
        Assert.assertEquals(bs.balanceCount("AA"), 0);

        /*------ m == n == distinctChars ------*/
```

```
        Assert.assertEquals(bs.balanceCount(mndBalanced), 0); // no n

        char lastChar = mndBalanced.charAt(mndBalanced.length() - 1);
        String mndOneDup = mndBalanced.replace(lastChar, (char) (last
(distinctChars -2) chars occur exactly once, one occurs twice, one is
one step away to balance,
        Assert.assertEquals(mndOneDup.length(), bs.getDistinctChars()
        Assert.assertEquals(bs.balanceCount(mndOneDup), 1); // just r
char with missing char,
    }

    // invalid input,
    @Test(expectedExceptions = IllegalArgumentException.class)
    public void testInvalidInput() {
        Assert.assertEquals(bs.balanceCount("ABAc"), 1);
    }

    // invalid char range, for constructor,
    @Test(expectedExceptions = IllegalArgumentException.class)
    public void testInvalidRange() {
        new BalanceString('z', 'a');
    }

    /**
     * Generate a string, with first char occur once, second twice, t
and so on.
     * <p>e.g A, ABB, ABBCCC, ABBCCCDDDD,
     *
     * @param m distinct char count,
     * @return
     */
    private String genIncSeq(int m) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < m; i++) {
            for (int j = 0; j <= i; j++) sb.append((char) (bs.getMinC
        }
        return sb.toString();
    }

    /**
     * Generate a string that contains each possible char exactly onc
     * <p>For [A-Z], it could be: "ABCDEFGHIJKLMNOPQRSTUVWXYZ".
     *
     * @return
```

```
            char minChar = bs.getMinChar();
            int distinctChars = bs.getDistinctChars();

            for (int i = 0; i < distinctChars; i++) {
                sb.append((char) (minChar + i));
            }
            return sb.toString();
        }
    }
```

All test cases would pass.

Complexity

- Time: `O(len) + O(m * lg(m)) + O(m * factorCount)`
  - Each sequential scan takes `O(len)`, there are several sequential loop.
  - Sorting of array takes `O(m*lg(m))`, which is at most `O(distinctChars * lg(distinctChars))`, thus constant, and only sort once.
  - To figure out moves for each n, takes `O(m)`.
  - The count of n that needs to figure moves, depends on count of divisible numbers for len, in the range [`minChar`, `maxChar`].
    This count if kind small & constant too.
- Space: `O(len)`
  - Input string need `O(len)`.
  - Counter hashmap need `O(m)`.
  - Sorted occurrence array need `O(m)`.

Where:

- `len` is string length.

- `maxN` max possible distinct char count, included,
- `factorCount` divisible number count in range `[1, n]`, by `len`,
- `minChar` min char, e.g A
- `maxChar` max char, e.g Z

And:

- `len >= m`
- `m <= distinctChars`

edited 2 days ago

ruakh
125k 13 201 256

answered Feb 2 at 16:30

Eric Wang
8,263 5 67 103

This answer is wrong. The only real difference from my algorithm is an optimization where you limit the range of numbers of distinct letters that you consider: instead of considering everything from 1 distinct letter to 26 distinct letters, you just consider the nearest divisors of the number of total letters. The problem is that sometimes the best solution isn't from one of those divisors, so that optimization means your code gives the wrong answer. *[continued]* – ruakh Feb 2 at 20:15

[continued] For example: for `ABBBBB`, your code gives `2` instead of `1`, because it only tries as few as 2 distinct letters, so it chooses `AAABBB` instead of `BBBBBB`. Similarly, for `ABCCCCCC`, your code gives `3` instead of `2` (`AAAACCCC` instead of `CCCCCCCC`). – ruakh Feb 2 at 20:17

@ruakh Thanks for the correction, I have refactor the code, and updated the answer accordingly, the new code actually becomes simpler with 2 functions removed, and I also generalized it to accept any valid char set, with a default instance as A-Z. – Eric Wang Feb 3 at 8:24