Minimum Unique Word Abbreviation

A string such as `"word"` contains the following abbreviations:

```
["word", "1ord", "w1rd", "wo1d", "wor1", "2rd", "w2d", "wo2", "1o1d", "1or1", "w1r1
", "1o2", "2r1", "3d", "w3", "4"]
```

Given a target string and a set of strings in a dictionary, find an abbreviation of this target string with the ***smallest possible*** length such that it does not conflict with abbreviations of the strings in the dictionary.

Each **number** or letter in the abbreviation is considered length = 1. For example, the abbreviation "a32bc" has length = 4.

Note:

- In the case of multiple answers as shown in the second example below, you may return any one of them.
- Assume length of target string = **m**, and dictionary size = **n**. You may assume that $\mathbf{m \le 21}$, $\mathbf{n \le 1000}$, and $\mathbf{\log_2(n) + m \le 20}$.

Examples:

```
"apple", ["blade"] -> "a4" (because "5" or "4e" conflicts with "blade")

"apple", ["plain", "amber", "blade"] -> "1p3" (other valid answers include "ap3", "
a3e", "2p2", "3le", "3l1").
```

Solution 1

I think this problem is NP-hard. For proof, I'll reduce the**set cover** decision problem to it. The example there is:

```
U = {1, 2, 3, 4, 5}
S = {{1, 2, 3}, {2, 4}, {3, 4}, {4, 5}}
```

Now I'd encode that for **Minimum Unique Word Abbreviation** like this:

```
target: "oooo"
dict:   "looo" (word 1)
        "lloo" (word 2)
        "lolo" (word 3)
        "olll" (word 4)
        "oool" (word 5)
```

The five dictionary words correspond to the five elements of U. And every word has four letters, corresponding to the four subsets in S. You can see for example the second subset {2, 4} encoded as the second colum in the dictionary, which has `l` in words 2 and 4 (and otherwise only `o`).

(Side note: You can build an abbreviation by picking letters and then replacing the unpicked ones with numbers. For example, when you have `leetcode` and pick the letters `l`, `t` and the last `e`, you get `l2t3e`. This way you can get all $2^{|word|}$ possible abbreviations from the $2^{|word|}$ possible ways of picking, as I've done in the first solution here.)

Now an optimal abbreviation for the target is "o2o", which we get by picking the first "o" because that distinguishes the target from words {1, 2, 3} and by picking the last "o" because that distinguishes the target from words {4, 5}. Just like picking the subsets {1, 2, 3} and {4, 5} cover all of U in the set cover problem.

A little problem is that if I switch the third and fourth element of S, then an optimal abbreviation picks the first and *third* "o", so instead of "o2o" we'd get "o1o1". That's a different length. So instead of the above encoding, I'd really do this, inserting "x" before and after each letter:

```
target: "xoxoxoxox"
dict:   "xlxoxoxox"
        "xlxlxoxox"
        "xlxoxlxox"
        "xoxlxlxlx"
        "xoxoxoxlx"
```

Now "1o5o1" is an optimal abbreviation, and when I switch third and fourth element of S again, I instead get "1o3o3", which still has the same length.

To be precise and usable: When I pick k letters, the optimal abbreviation will have those k letters as well as k+1 numbers, so length 2k+1. That means minimizing the abbreviation length is equivalent to minimizing the number of picked letters, which is equivalent to minimizing the number of picked elements of S in the decision problem.

Summary: To solve a given **set cover** decision problem instance, I translate it to a **Minimum Unique Word Abbreviation** instance like above, and then from the computed optimal abbreviation, I subtract 1 from its length and divide it by 2 to get the number of picked letters, which is the minimum number of elements of S that cover U. And with that, I can directly answer the original set cover instance.

Moral of the story: Don't feel bad about writing a brute force solution :-)

written by StefanPochmann original link here

Solution 2

The key idea of my solution is to preprocess the dictionary to transfer all the words to bit sequences (int):
Pick the words with same length as target string from the dictionary and compare the characters with target. If the characters are different, set the corresponding bit to 1, otherwise, set to 0.
Ex: "abcde", ["abxdx", "xbcdx"] => [00101, 10001]

The problem is now converted to find a bit mask that can represent the shortest abbreviation, so that for all the bit sequences in dictionary, mask & bit sequence > 0.
Ex: for [00101, 10001], the mask should be [00001]. if we mask the target string with it, we get "****e" ("4e"), which is the abbreviation we are looking for.

To find the bit mask, we need to perform DFS with some optimizations. But which bits should be checked? We can perform "or" operation for all the bit sequences in the dictionary and do DFS for the "1" bits in the result.
Ex: 00101 | 10001 = 10101, so we only need to take care of the 1st, 3rd, and 5th bit.

Here is a C++ implementation, the running time should be about 3ms. Any suggestions would be appreciated.

```
class Solution {
    int n, cand, bn, minlen, minab;
    vector<int> dict;

    // Return the length of abbreviation given bit sequence
    int abbrLen(int mask) {
        int count = 0;
        for (int b = 1; b < bn;) {
            if ((mask & b) == 0)
                for (; b < bn and (mask & b) == 0; b <<= 1);
            else b <<= 1;
            count ++;
        }
        return count;
    }

    // DFS backtracking
    void dfs(int bit, int mask) {
        int len = abbrLen(mask);
        if (len >= minlen) return;
        bool match = true;
        for (auto d : dict) {
            if ((mask & d) == 0) {
                match = false;
                break;
            }
        }
        if (match) {
            minlen = len;
            minab = mask;
        }
        else
            for (int b = bit; b < bn; b <<= 1)
            for (int b = bit; b < bn; b <<= 1)
                if (cand & b) dfs(b << 1, mask + b);
    }

public:
    string minAbbreviation(string target, vector<string>& dictionary) {
        n = target.size(), bn = 1 << n, cand = 0, minlen = INT_MAX;
        string res;

        // Preprocessing with bit manipulation
        for (auto w : dictionary) {
            int word = 0;
            if (w.size() != n) continue;
            for (int i = n-1, bit = 1; i >= 0; --i, bit <<= 1)
                if (target[i] != w[i]) word += bit;
            dict.push_back(word);
            cand |= word;
        }
        dfs(1, 0);

        // Reconstruct abbreviation from bit sequence
        for (int i = n-1, pre = i; i >= 0; --i, minab >>= 1) {
            if (minab & 1) {
                if (pre-i > 0) res = to_string(pre-i) + res;
                pre = i - 1;
                res = target[i] + res;
            }
            else if (i == 0) res = to_string(pre-i+1) + res;
        }
        return res;
    }
};
```

UPDATE: a better way to determine the length of abbreviation mentioned by @StefanPochmann

```
int abbrLen(int mask) {
    int count = n;
    for (int b = 3; b < bn; b <<= 1)
        if ((mask & b) == 0)
            count --;
    return count;
}
```

written by topcoder007 original link here

Solution 3

Gets accepted in ~130 ms.

```
def minAbbreviation(self, target, dictionary):
    m = len(target)
    diffs = {sum(2**i for i, c in enumerate(word) if target[i] != c)
             for word in dictionary if len(word) == m}
    if not diffs:
        return str(m)
    bits = max((i for i in range(2**m) if all(d & i for d in diffs)),
               key=lambda bits: sum((bits >> i) & 3 == 0 for i in range(m-1)))
    s = ''.join(target[i] if bits & 2**i else '#' for i in range(m))
    return re.sub('#+', lambda m: str(len(m.group())), s)
```

If the target is `apple` and the dictionary contains `apply`, then the abbreviation must include the `e` as the letter `e`, not in a number. It's the only letter distinguishing these two words. Similarly, if the dictionary contains `tuple`, then the abbreviation must include the `a` or the first `p` as a letter.

For each dictionary word (of correct size), I create a diff-number whose bits tell me which of the word's letters differ from the target. Then I go through the 2^m possible abbreviations, represented as number from 0 to 2^m-1, the bits representing which letters of target are in the abbreviation. An abbreviation is ok if it doesn't match any dictionary word. To check whether an abbreviation doesn't match a dictionary word, I simply check whether the abbreviation number and the dictionary word's diff-number have a common 1-bit. Which means that the abbreviation contains a letter where the dictionary word differs from the target.

Then from the ok abbreviations I find one that maximizes how much length it saves me. Two consecutive 0-bits in the abbreviation number mean that the two corresponding letters will be encoded as the number 2. It saves length 1. Three consecutive 0-bits save length 2, and so on. To compute the saved length, I just count how many pairs of adjacent bits are zero.

Now that I have the number representing an optimal abbreviation, I just need to turn it into the actual abbreviation. First I turn it into a string where each 1-bit is turned into the corresponding letter of the target and each 0-bit is turned into `#`. Then I replace streaks of `#` into numbers.

written by StefanPochmann original link here