Word Squares

Given a set of words **(without duplicates)**, find all word squares you can build from them.

A sequence of words forms a valid word square if the k^{th} row and column read the exact same string, where $0 \le k <$ max(numRows, numColumns).

For example, the word sequence `["ball","area","lead","lady"]` forms a word square because each word reads the same both horizontally and vertically.

```
b a l l
a r e a
l e a d
l a d y
```

Note:

1. There are at least 1 and at most 1000 words.
2. All words will have the exact same length.
3. Word length is at least 1 and at most 5.
4. Each word contains only lowercase English alphabet `a-z`.

Example 1:

```
Input:
["area","lead","wall","lady","ball"]

Output:
[
  [ "wall",
    "area",
    "lead",
    "lady"
  ],
  [ "ball",
    "area",
    "lead",
    "lady"
  ]
]

Explanation:
The output consists of two word squares. The order of output does not matter (just
the order of words in each word square matters).
```

Example 2:

Input:
["abat","baba","atan","atal"]

Output:
```
[
  [ "baba",
    "abat",
    "baba",
    "atan"
  ],
  [ "baba",
    "abat",
    "baba",
    "atal"
  ]
]
```

Explanation:
The output consists of two word squares. The order of output does not matter (just the order of words in each word square matters).

Solution 1

My first approach is brute-force, try every possible word sequences, and use the solution of Problem 422 (https://leetcode.com/problems/valid-word-square/) to check each sequence. This solution is straightforward, but too slow (TLE).

A better approach is to check the validity of the word square while we build it.
Example: `["area","lead","wall","lady","ball"]`
We know that the sequence contains 4 words because the length of each word is 4.
Every word can be the first word of the sequence, let's take `"wall"` for example.
Which word could be the second word? Must be a word start with `"a"` (therefore `"area"`), because it has to match the second letter of word `"wall"`.
Which word could be the third word? Must be a word start with `"le"` (therefore `"lead"`), because it has to match the third letter of word `"wall"` and the third letter of word `"area"`.
What about the last word? Must be a word start with `"lad"` (therefore `"lady"`).
For the same reason above.

The picture below shows how the prefix are matched while building the sequence.

In order for this to work, we need to fast retrieve all the words with a given**prefix**.
There could be 2 ways doing this:

1. Using a hashtable, key is **prefix**, value is a list of words with that prefix.
2. Trie, we store a list of words with the**prefix** on each trie node.

The implemented below uses Trie.

```
public class Solution {
    class TrieNode {
        List<String> startWith;
        TrieNode[] children;

        TrieNode() {
            startWith = new ArrayList<>();
            children = new TrieNode[26];
        }
    }

    class Trie {
        TrieNode root;

        Trie(String[] words) {
            root = new TrieNode();
            for (String w : words) {
                TrieNode cur = root;
                for (char ch : w.toCharArray()) {
                    int idx = ch - 'a';
                    if (cur.children[idx] == null)
                        cur.children[idx] = new TrieNode();
                    cur.children[idx].startWith.add(w);
                    cur = cur.children[idx];
```

```
                }
            }
        }

        List<String> findByPrefix(String prefix) {
            List<String> ans = new ArrayList<>();
            TrieNode cur = root;
            for (char ch : prefix.toCharArray()) {
                int idx = ch - 'a';
                if (cur.children[idx] == null)
                    return ans;

                cur = cur.children[idx];
            }
            ans.addAll(cur.startWith);
            return ans;
        }
    }

    public List<List<String>> wordSquares(String[] words) {
        List<List<String>> ans = new ArrayList<>();
        if (words == null || words.length == 0)
            return ans;
        int len = words[0].length();
        Trie trie = new Trie(words);
        List<String> ansBuilder = new ArrayList<>();
        for (String w : words) {
            ansBuilder.add(w);
            search(len, trie, ans, ansBuilder);
            ansBuilder.remove(ansBuilder.size() - 1);
        }

        return ans;
    }

    private void search(int len, Trie tr, List<List<String>> ans,
            List<String> ansBuilder) {
        if (ansBuilder.size() == len) {
            ans.add(new ArrayList<>(ansBuilder));
            return;
        }

        int idx = ansBuilder.size();
        StringBuilder prefixBuilder = new StringBuilder();
        for (String s : ansBuilder)
            prefixBuilder.append(s.charAt(idx));
        List<String> startWith = tr.findByPrefix(prefixBuilder.toString());
        for (String sw : startWith) {
            ansBuilder.add(sw);
            search(len, tr, ans, ansBuilder);
            ansBuilder.remove(ansBuilder.size() - 1);
        }
    }
}
```

written by lzb700m original link here

Solution 2

Python Solution (accepted in ~870 ms)

```
def wordSquares(self, words):
    n = len(words[0])
    fulls = collections.defaultdict(list)
    for word in words:
        for i in range(n):
            fulls[word[:i]].append(word)
    def build(square):
        if len(square) == n:
            squares.append(square)
            return
        for word in fulls[''.join(zip(*square)[len(square)])]:
            build(square + [word])
    squares = []
    for word in words:
        build([word])
    return squares
```

Explanation

I try every word for the first row. For each of them, try every fitting word for the second row. And so on. The first few rows determine the first few columns and thus determine how the next row's word must start. For example:

```
wall      Try words      wall                    wall                      wall
a...   => starting  =>   area      Try words     area                      area
l...      with "a"       le..   => starting  =>  lead      Try words       lead
l...                     la..      with "le"     lad.   => starting   =>  lady
                                                          with "lad"
```

For quick lookup, my `fulls` dictionary maps prefixes to lists of words who have that prefix.

C++ Solution (accepted in ~180 ms)

```
class Solution {
public:
    vector<vector<string>> wordSquares(vector<string>& words) {
        n = words[0].size();
        square.resize(n);
        for (string word : words)
            for (int i=0; i<n; i++)
                fulls[word.substr(0, i)].push_back(word);
        build(0);
        return squares;

    }
    int n;
    unordered_map<string, vector<string>> fulls;
    vector<string> square;
    vector<vector<string>> squares;
    void build(int i) {
        if (i == n) {
            squares.push_back(square);
            return;
        }
        string prefix;
        for (int k=0; k<i; k++)
            prefix += square[k][i];
        for (string word : fulls[prefix]) {
            square[i] = word;
            build(i + 1);
        }
    }
};
```

written by StefanPochmann original link here

Solution 3

Optimized to 70ms by pre-size the vec instead of push_back and pop_back

```cpp
class Solution {
public:
    struct TrieNode {
        vector<int> indexs;
        vector<TrieNode*> children;
        TrieNode() {
            children.resize(26, nullptr);
        }
    };

    TrieNode* buildTrie(vector<string>& words) {
        TrieNode* root = new TrieNode();
        for (int j = 0; j < words.size(); j++) {
            TrieNode* t = root;
            for (int i = 0; i < words[j].size(); i++) {
                if (!t->children[words[j][i] - 'a'])
                    t->children[words[j][i] - 'a'] = new TrieNode();
                t = t->children[words[j][i] - 'a'];
                t->indexs.push_back(j);
            }
        }
        return root;
    }

    vector<vector<string>> res;
    vector<string> vec;
    void backtrack(vector<string>& words, int level, TrieNode* root) {
        if (level >= words[0].size()) {
            res.push_back(vec);
            return;
        }
        string str = "";
        for (int i = 0; i < level; i++)
            str += vec[i][level];
        TrieNode* t = root;
        for (int i = 0; i < str.size(); i++)
            if (!(t = t->children[str[i] - 'a'])) return;
        for (auto index : t->indexs) {
            vec[level] = words[index];
            backtrack(words, level + 1, root);
        }
    }

    vector<vector<string>> wordSquares(vector<string>& words) {
        if (words.empty()) return res;
        TrieNode* root = buildTrie(words);
        vec.resize((int)words[0].size());
        for (auto& word : words) {
            vec[0] = word;
            backtrack(words, 1, root);
        }
        return res;
    }
};
```

written by zyoppy008 original link here

From Leetcoder.