Prefix and Suffix Search

Given many `words`, `words[i]` has weight `i`.

Design a class `WordFilter` that supports one function, `WordFilter.f(String prefix, String suffix)`. It will return the word with given `prefix` and `suffix` with maximum weight. If no word exists, return -1.

Examples:

```
Input:
WordFilter(["apple"])
WordFilter.f("a", "e") // returns 0
WordFilter.f("b", "") // returns -1
```

Note:

1. `words` has length in range `[1, 15000]`.
2. For each test case, up to `words.length` queries `WordFilter.f` may be made.
3. `words[i]` has length in range `[1, 10]`.
4. `prefix, suffix` have lengths in range `[0, 10]`.
5. `words[i]` and `prefix, suffix` queries consist of lowercase letters only.

Solution 1

Before solving this problem, we need to know which operation is called the most.

If f is more frequently than WordFilter, use method 1.
If space complexity is concerned, use method 2.
If the input string array might change frequently, use method 3.

< Method 1 >
WordFilter: Time = O(NL^2)
f: Time = O(1)
Space = O(NL^2)
Note: N is the size of input array and L is the max length of input strings.

```
class WordFilter {
    HashMap<String, Integer> map = new HashMap<>();

    public WordFilter(String[] words) {
        for(int w = 0; w < words.length; w++){
            for(int i = 0; i <= 10 && i <= words[w].length(); i++){
                for(int j = 0; j <= 10 && j <= words[w].length(); j++){
                    map.put(words[w].substring(0, i) + "#" + words[w].substring(wor
ds[w].length()-j), w);
                }
            }
        }
    }

    public int f(String prefix, String suffix) {
        return (map.containsKey(prefix + "#" + suffix))? map.get(prefix + "#" + suff
ix) : -1;
    }
}
```

< Method 2 >
WordFilter: Time = O(NL)
f: Time = O(N)
Space = O(NL)

```
class WordFilter {
    HashMap<String, List<Integer>> mapPrefix = new HashMap<>();
    HashMap<String, List<Integer>> mapSuffix = new HashMap<>();

    public WordFilter(String[] words) {

        for(int w = 0; w < words.length; w++){
            for(int i = 0; i <= 10 && i <= words[w].length(); i++){
                String s = words[w].substring(0, i);
                if(!mapPrefix.containsKey(s)) mapPrefix.put(s, new ArrayList<>());
                mapPrefix.get(s).add(w);
            }
            for(int i = 0; i <= 10 && i <= words[w].length(); i++){
                String s = words[w].substring(words[w].length() - i);
                if(!mapSuffix.containsKey(s)) mapSuffix.put(s, new ArrayList<>());
                mapSuffix.get(s).add(w);
            }
        }

    public int f(String prefix, String suffix) {

        if(!mapPrefix.containsKey(prefix) || !mapSuffix.containsKey(suffix)) return
-1;
        List<Integer> p = mapPrefix.get(prefix);
        List<Integer> s = mapSuffix.get(suffix);
        int i = p.size()-1, j = s.size()-1;
        while(i >= 0 && j >= 0){
            if(p.get(i) < s.get(j)) j--;
            else if(p.get(i) > s.get(j)) i--;
            else return p.get(i);
        }
        return -1;
```

< Method 3 >

WordFilter: Time = O(1)
f: Time = O(NL)
Space = O(1)

```
class WordFilter {
    String[] input;
    public WordFilter(String[] words) {
        input = words;
    }
    public int f(String prefix, String suffix) {
        for(int i = input.length-1; i >= 0; i--){
            if(input[i].startsWith(prefix) && input[i].endsWith(suffix)) return i;
        }
        return -1;
    }
}
```

written by moto72 original link here

Solution 2

Hello,

May I ask what does the following test case mean?

```
Input:
[["apple"]], ["a","e"]]
Expected:
[null,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,-1,-1,0,0,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1]
```

I think it would be using words ["apple"] to initialize the word filter,
then find word with prefix "a" and suffix "e".
But why is the expected result so long a list?

Thanks in advance.

written by songzy12 original link here

Solution 3

This is easy to understand.

```
struct Trie {
    vector<int> words; // index of words
    vector<Trie *> children;
    Trie() {
        children = vector<Trie *>(26, nullptr);
    }
    // Thanks to @huahualeetcode for adding this in case of memory leak
    ~Trie() {
        for (int i = 0; i < 26; i++) {
            if (children[i]) {
                delete children[i];
            }
        }
    }

    void add(const string &word, size_t begin, int index) {
        words.push_back(index);
        if (begin < word.length()) {
            if (!children[word[begin] - 'a']) {
                children[word[begin] - 'a'] = new Trie();
            }
            children[word[begin] - 'a']->add(word, begin + 1, index);
        }
    }

    vector<int> find(const string &prefix, size_t begin) {
        if (begin == prefix.length()) {
            return words;
        } else {
            if (!children[prefix[begin] - 'a']) {
                return {};
            } else {
                return children[prefix[begin] - 'a']->find(prefix, begin + 1);
            }
        }
    }
};

class WordFilter {
public:
    WordFilter(vector<string> words) {
        forwardTrie = new Trie();
        backwardTrie = new Trie();
        for (int i = 0; i < words.size(); i++) {
            string word = words[i];
            forwardTrie->add(word, 0, i);
            reverse(word.begin(), word.end());
            backwardTrie->add(word, 0, i);
        }
    }

    int f(string prefix, string suffix) {
        auto forwardMatch = forwardTrie->find(prefix, 0);
```

```
        reverse(suffix.begin(), suffix.end());
        auto backwardMatch = backwardTrie->find(suffix, 0);
        // search from the back
        auto fIter = forwardMatch.rbegin(), bIter = backwardMatch.rbegin();
        while (fIter != forwardMatch.rend() && bIter != backwardMatch.rend()) {
            if (*fIter == *bIter) {
                return *fIter;
            } else if (*fIter > *bIter) {
                fIter ++;
            } else {
                bIter ++;
            }
        }
        return -1;
    }

private:
    Trie *forwardTrie, *backwardTrie;
};
```

written by Frank_Fan original link here