Number of Matching Subsequences

Given string `S` and a dictionary of words `words`, find the number of `words[i]` that is a subsequence of `S`.

Example :
Input:
`S = "abcde"`
`words = ["a", "bb", "acd", "ace"]`
Output: 3
Explanation: There are three words in `words` that are a subsequence of `S`: "a", "acd", "ace".

Note:

- All words in `words` and `S` will only consists of lowercase letters.
- The length of `S` will be in the range of `[1, 50000]`.
- The length of `words` will be in the range of `[1, 5000]`.
- The length of `words[i]` will be in the range of `[1, 50]`.

Solution 1

The idea is to use the String S and build a dictionary of indexes for each character in the string.
Then for each word, we can go character by character and see if it is a subsequence, by using the corresponding index dictionary, but just making sure that the index of the current character occurs after the index where the previous character was seen.
To speed up the processing, we should use binary search in the index dictionary.

As an example for S = "aacbacbde"
the
dict_idxs =
{a: [0, 1, 4]
b: [3, 6]
c: [2, 5]
d: [7]
e: [8]
}
Now for the word say “abcb”, starting with d_i = 0,
a => get list for a in the dict_idxs which is [0, 1, 4], and in the list, find the index of a >= d_i which is 0. After this update d_i to +1 => 1
b => get list for b in the dict_idxs [3, 6], and in the list, find the index of b >= d_i => >= 1, which is at index 0 => 3, after this update d_i to 4+1 => 4
c => in the list for c, [3, 5], find the index of c >= 4, which is 5. update d_i to 5+1 = 6
b => in the list for b, [3, 6], fund the index of b >= 6, which is 6, update d_i to 7,
since this is the end of the word, and we have found all characters in the word in the index dictionary, we return True for this word.

```
    def numMatchingSubseq(self, S, words):
        def isMatch(word, w_i, d_i):
            if w_i == len(word): return True
            l = dict_idxs[word[w_i]]
            if len(l) == 0 or d_i > l[-1]: return False
            i = l[bisect_left(l, d_i)]
            return isMatch(word, w_i + 1, i + 1)

        dict_idxs = defaultdict(list)
        for i in range(len(S)):
            dict_idxs[S[i]].append(i)
        return sum(isMatch(word, 0, 0) for word in words)
```

written by johnyrufus16 original link here

Solution 2

Runtime is linear in the total size of the input (`S` and all of `words`). Explanation below the code.

Python:

```python
def numMatchingSubseq(self, S, words):
    waiting = collections.defaultdict(list)
    for w in words:
        waiting[w[0]].append(iter(w[1:]))
    for c in S:
        for it in waiting.pop(c, ()):
            waiting[next(it, None)].append(it)
    return len(waiting[None])
```

C++:

```cpp
int numMatchingSubseq(string S, vector<string>& words) {
    vector<pair<int, int>> waiting[128];
    for (int i = 0; i < words.size(); i++)
        waiting[words[i][0]].emplace_back(i, 1);
    for (char c : S) {
        auto advance = waiting[c];
        waiting[c].clear();
        for (auto it : advance) {
            int i = it.first, j = it.second;
            waiting[words[i][j]].emplace_back(i, j + 1);
        }
    }
    return waiting[0].size();
}
```

Java:

```java
public int numMatchingSubseq(String S, String[] words) {
    List<Integer[]>[] waiting = new List[128];
    for (int c = 0; c <= 'z'; c++)
        waiting[c] = new ArrayList();
    for (int i = 0; i < words.length; i++)
        waiting[words[i].charAt(0)].add(new Integer[]{i, 1});
    for (char c : S.toCharArray()) {
        List<Integer[]> advance = new ArrayList(waiting[c]);
        waiting[c].clear();
        for (Integer[] a : advance)
            waiting[a[1] < words[a[0]].length() ? words[a[0]].charAt(a[1]++) : 0].a
dd(a);
    }
    return waiting[0].size();
}
```

Explanation:

For each letter I remember the words currently waiting for that letter. For example:

```
S = "abcde"
words = ["a", "bb", "acd", "ace"]
```

I store that `"a"`, `"acd"` and `"ace"` are waiting for an `'a'` and `"bb"` is waiting for a `'b'` (using parentheses to show how far I am in each word):

```
'a':  ["(a)", "(a)cd", "(a)ce"]
'b':  ["(b)b"]
```

Then I go through S. First I see `'a'`, so I take the list of words waiting for `'a'` and store them as waiting under their next letter:

```
'b':  ["(b)b"]
'c':  ["a(c)d", "a(c)e"]
None: ["a"]
```

You see `"a"` is already waiting for nothing anymore, while `"acd"` and `"ace"` are now waiting for `'c'`. Next I see `'b'` and update accordingly:

```
'b':  ["b(b)"]
'c':  ["a(c)d", "a(c)e"]
None: ["a"]
```

Then `'c'`:

```
'b':  ["b(b)"]
'd':  ["ac(d)"]
'e':  ["ac(e)"]
None: ["a"]
```

Then `'d'`:

```
'b':  ["b(b)"]
'e':  ["ac(e)"]
None: ["a", "acd"]
```

Then `'e'`:

```
'b':  ["b(b)"]
None: ["a", "acd", "ace"]
```

And now I just return how many words aren't waiting for anything anymore.

Implementation Details:

In Python I use iterators to represent words and their progress. In C++/Java I use pairs (i,j) which tell me that it's word i and I'm at letter j. In Go (below) I just keep slicing off letters from the front of the strings. For C++ I also tried `istringstream` but failed, the compiler gave me the typical cryptic compile error wall.

Variations:

Cleaner initialization, using iterators iterating the whole word instead of a copy of the word after its first letter:

```
def numMatchingSubseq(self, S, words):
    waiting = collections.defaultdict(list)
    for it in map(iter, words):
        waiting[next(it)].append(it)
    for c in S:
        for it in waiting.pop(c, ()):
            waiting[next(it, None)].append(it)
    return len(waiting[None])
```

Make all words waiting for a space character and prepend a space character to S:

```
def numMatchingSubseq(self, S, words):
    waiting = collections.defaultdict(list, {' ': map(iter, words)})
    for c in ' ' + S:
        for it in waiting.pop(c, ()):
            waiting[next(it, None)].append(it)
    return len(waiting[None])
```

This avoids some code duplication and now the solution can also handle empty words.

A Go version:

```
func numMatchingSubseq(S string, words []string) (num int) {
    waiting := map[rune][]string{' ': words}
    for _, c := range " " + S {
        advance := waiting[c]
        delete(waiting, c)
        for _, word := range advance {
            if len(word) == 0 {
                num++
            } else {
                waiting[rune(word[0])] = append(waiting[rune(word[0])], word[1:])
            }
        }
    }
    return
}
```

written by StefanPochmann original link here

Solution 3

```
class Solution {
    public int numMatchingSubseq(String S, String[] words) {
        Map<Character, Deque<String>> map = new HashMap<>();
        for (char c = 'a'; c <= 'z'; c++) {
            map.putIfAbsent(c, new LinkedList<String>());
        }
        for (String word : words) {
            map.get(word.charAt(0)).addLast(word);
        }

        int count = 0;
        for (char c : S.toCharArray()) {
            Deque<String> queue = map.get(c);
            int size = queue.size();
            for (int i = 0; i < size; i++) {
                String word = queue.removeFirst();
                if (word.length() == 1) {
                    count++;
                } else {
                    map.get(word.charAt(1)).addLast(word.substring(1));
                }
            }
        }
        return count;
    }
}
```

substring() is time consuming and you could also use an array as a map to further improve this solution.

written by setsuna214 original link here