<https://leetcode.com/problems/number-of-matching-subsequences/description/>

Given a string s and an array of strings words, return the number of words[i] that is a subsequence of s.

A **subsequence** of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters.

For example, "ace" is a subsequence of "abcde".

**Example 1:**

**Input:** s = "abcde", words = ["a","bb","acd","ace"]

**Output:** 3

**Explanation:** There are three strings in words that are a subsequence of s: "a", "acd", "ace".

**Example 2:**

**Input:** s = "dsahjpjauf", words = ["ahjpjau","ja","ahbwzgqnuk","tnmlanowax"]

**Output:** 2

**Constraints:**

1 <= s.length <= 5 \* 10^4

1 <= words.length <= 5000

1 <= words[i].length <= 50

s and words[i] consist of only lowercase English letters.

**Attempt 1: 2024-12-31**

**Solution 1: Queue like Revolver + HashMap (120 min)**

class Solution {

    public int numMatchingSubseq(String s, String[] words) {

        // Create a map to store a queue of words waiting for each character in `s`

        Map<Character, Queue<String>> map = new HashMap<>();

        // Initialize the map with all characters from the string `s`

        for (int i = 0; i < s.length(); i++) {

            map.putIfAbsent(s.charAt(i), new LinkedList<>());

        }

        // Group words by their starting character

        for (String word : words) {

            char startChar = word.charAt(0); // The first character of the word

            // If the character exists in `s`, add the word to the respective queue

            if (map.containsKey(startChar)) {

                map.get(startChar).add(word);

            }

        }

        // Count of matching subsequences

        int count = 0;

        // Process each character in 's' like a revolver

        for (int i = 0; i < s.length(); i++) {

            // Current character in 's'

            char startChar = s.charAt(i);

            // Queue of words waiting for this character to continue process like revolver

            Queue<String> q = map.get(startChar);

            // Process each word one by one

            int size = q.size();

            for (int j = 0; j < size; j++) {

                String str = q.poll();

                // If the word is fully matched(remain last char still match the current

                // character from 's'), increment the count means one subsequence find

                if (str.length() == 1) {

                    count++;

                // Otherwise, move the remain word(remove 1st char) to the corresponding

                // queue based on its next character

                } else {

                    // The next character of current string will be the key for adding

                    // back to map

                    char newStartChar = str.charAt(1);

                    // Only move if map contains this key

                    if (map.containsKey(newStartChar)) {

                        map.get(newStartChar).add(str.substring(1));

                    }

                }

            }

        }

        return count;

    }

}

Time Complexity: O(n + w), where n is the length of s, w is the total number of characters in words.

Space Complexity: O(n + w)

**Explanation of Logic:**

Initialization of Map:

The map is initialized to contain a queue for every character in the string s.

Each queue will store words (or remaining portions of words) that are waiting for the corresponding character to match.

Grouping Words by Starting Character:

Each word in words is added to the queue corresponding to its first character, provided that character exists in s.

Processing Characters in **s:**

For each character in s, process all words waiting for this character (i.e., in its queue).

For every word in the queue:

If the word is fully matched (length is 1), increment the count.

Otherwise, move the remaining portion of the word to the queue of its next required character.

Final Result:

The count variable accumulates the number of fully matched subsequences.

**Time Complexity:**

Initialization of Map: O(n), where n is the length of s.

Adding Words to Queues: O(w), where w is the total number of characters in words.

Processing Queues: Each character in words is processed once, so O(w).

Total: O(n + w).

**Space Complexity:**

Map and Queues: O(n + w), where:

O(n) is for the map entries corresponding to characters in s.

O(w) is for storing the words in the queues.

**Refer to**

<https://leetcode.com/problems/number-of-matching-subsequences/solutions/117634/efficient-and-simple-go-through-words-in-parallel-with-explanation/>

**Java:**

Using **StringCharacterIterator** (requires import java.text.StringCharacterIterator;):

public int numMatchingSubseq(String S, String[] words) {

List<StringCharacterIterator>[] waiting = new List[128];

for (int c = 0; c <= 'z'; c++)

waiting[c] = new ArrayList();

for (String w : words)

waiting[w.charAt(0)].add(new StringCharacterIterator(w));

for (char c : S.toCharArray()) {

List<StringCharacterIterator> advance = waiting[c];

waiting[c] = new ArrayList();

for (StringCharacterIterator it : advance)

waiting[it.next() % it.DONE].add(it);

}

return waiting[0].size();

}

Using index pairs (i,j) meaning word i is waiting for its letter j:

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].add(a);

}

return waiting[0].size();

}

**Explanation:**

I go through S once, and while I'm doing that, I move through all words accordingly. That is, I keep track of how much of each word I've already seen, and with each letter of S, I advance the words waiting for that letter. To quickly find the words waiting for a certain letter, I store each word (and its progress) in a list of words waiting for that letter. Then for each of the lucky words whose current letter just occurred in S, I update their progress and store them in the list for their next letter.

Let's go through the given 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.

**Refer to**

<https://leetcode.com/problems/number-of-matching-subsequences/solutions/2306523/step-by-step-explanation/>

**Step-By-Step Explanation**

The very bruteforce way to solve this problem is, we have the words of string. So, what we gonna do we iterate over the String s & if we find the subsequences (appearing of those characters in word string) then we gonna increment our count, otherwise if they are not present ignore them & carry on.

Let's look at the code,

class Solution {

public int numMatchingSubseq(String s, String[] words) {

int index = -1;

int count = 0;

boolean notSubSeq = true;

for(String word : words) {

for(char c : word.toCharArray()) {

index = s.indexOf(c, index + 1);

if (index == -1) {

notSubSeq = false;

break;

}

}

if(notSubSeq)

count++;

notSubSeq = true;

index = -1;

}

return count;

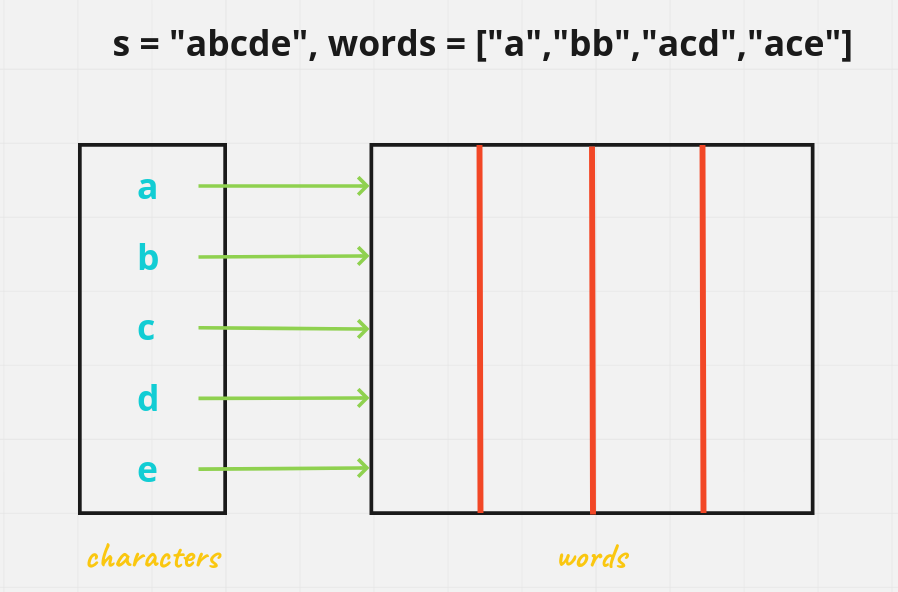
}

}

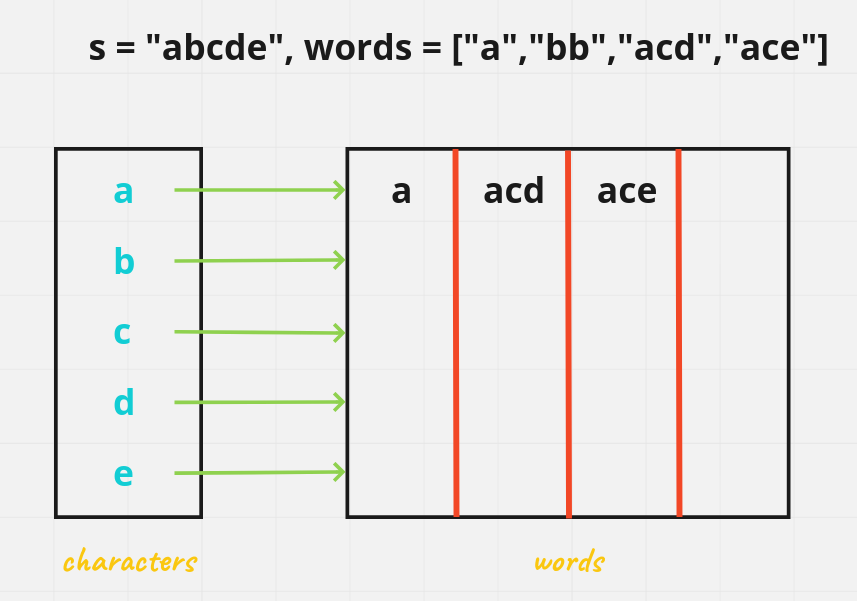
**Now, let's optimise it**

Input: s = "abcde", words = ["a","bb","acd","ace"]

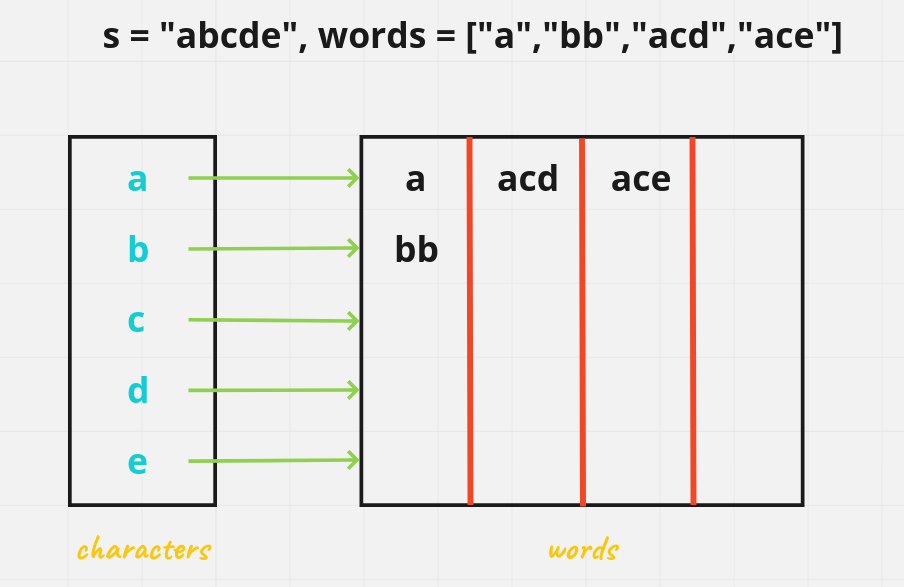
Output: 3



So, now first of all I'll gonna add all the words present in my array starting with characters a

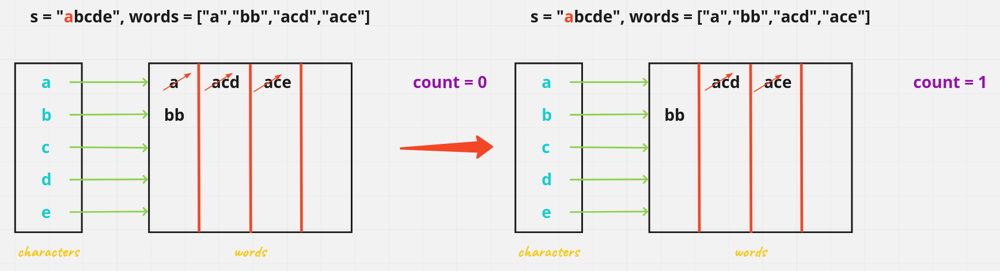


Next we have b characters to add



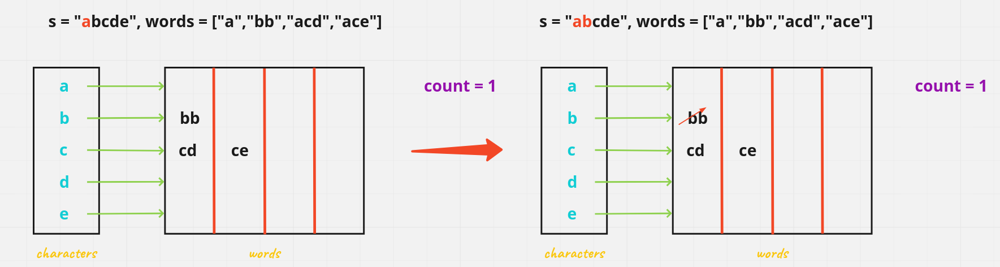
Now, what we'll do is iterate over all the characters present in our super-string s and remove their occurence from the map of that character.

So, intially we have **a** we gonna remove all the characters of **a** present in our word list and if any word becomes empty string we gonna increment our count



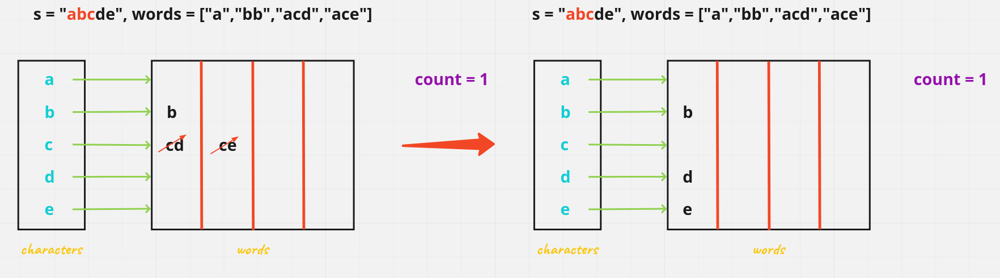
As, you can see as well we have "**a**" in our word dataset as well & our count is 1

So now what we do with remaining word present in front of **a** is delete from their entry and put them to new entry correspond to their characters, then what we gonna do is start our entry from next character in string i.e. **b** and remove all the characters present in our word array



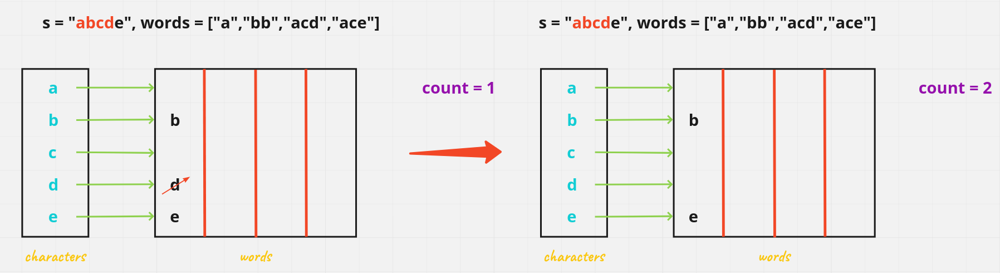
So now what we do with remaining word present in front of **b** is delete from their entry and put them to new entry correspond to their characters, but since none of the word become empty string yet, we won't increment our count.

Start our entry from next character in string i.e. **c** and remove all the characters present in our word array



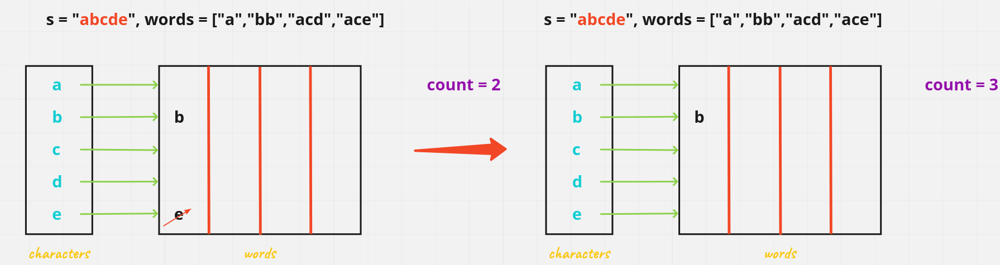
So now what we do with remaining word present in front of **c** is delete from their entry and put them to new entry correspond to their characters, but since none of the word become empty string yet, we won't increment our count.

Start our entry from next character in string i.e. **d** and remove all the characters present in our word array



As, our **d** character array wordlist becomes empty by removing the character from the string, so we gonna increment our count = 2

Start our entry from next character in string i.e. **e** and remove all the characters present in our word array



As, our **e** character array wordlist becomes empty by removing the character from the string, so we gonna increment our count = 3

And as you can see in the word there only is 3 word subsequences which are possible and our answer is 3 as well

words = ["a","acd","ace"]

Output: 3

**Let's code it up then**

class Solution {

public int numMatchingSubseq(String s, String[] words) {

Map<Character, Queue<String>> map = new HashMap<>();

int count = 0;

for(int i = 0; i < s.length(); i++) {

map.putIfAbsent(s.charAt(i), new LinkedList<>());

}

for(String word : words) {

char startingChar = word.charAt(0);

if(map.containsKey(startingChar)) {

map.get(startingChar).offer(word);

}

}

for(int i = 0; i < s.length(); i++) {

char startingChar = s.charAt(i);

Queue<String> q = map.get(startingChar);

int size = q.size();

for(int k = 0; k < size; k++) {

String str = q.poll();

if(str.substring(1).length() == 0) {

count++;

} else {

if(map.containsKey(str.charAt(1))) {

map.get(str.charAt(1)).add(str.substring(1));

}

}

}

}

return count;

}

}

**Solution 2: Trie (180 min, the convergence DFS logic quite similar to** [L1034.Coloring A Border (Ref.L200,L733)](note://WEB1a530db8c1e1bd0e15c9dff995bd2b72)**)**

**L792 对 Trie 的使用是个非典型的使用方案，一般都是在一堆字符串构建的 Trie 中寻找一个目标字符串，这里是在一堆字符串构建的 Trie 中将每个字符串再单独拎出来和另外一个模版字符串比较，每一次比较的时候 Trie 中存储的每个字符串上的字母投影在用来比较的另外一个模版字符串上的 index 并不是连续的，而是跳跃的，因为 Trie 中存储的字符串可能是该模版字符串的 subsequence**

class Solution {

    class TrieNode {

        char c;

        int findWordCount;

        TrieNode[] children;

        public TrieNode(char c) {

            this.c = c;

            this.findWordCount = 0;

            this.children = new TrieNode[26];

        }

    }

    public int numMatchingSubseq(String s, String[] words) {

        TrieNode root = createTrieTree(words);

        return helper(s, root, 0);

    }

    /\*\*

        e.g s = "abcde", words = ["a","bb","acd","ace"]

        There are three strings in words that are a

        subsequence of s: "a", "acd", "ace".

                                root.c = "\*"

                                root.findWordCount = 0

                                root.children

                            /                       \

                root.c = "a"                    root.c = "b"

                root.findWordCount = 1          root.findWordCount = 0

                root.children                   root.children

                        /                                 \

                root.c = "c"                    root.c = "b"

                root.findWordCount = 0          root.findWordCount = 1

                root.children                   root.children = null

                /                    \

        root.c = "d"              root.c = "e"

        root.findWordCount = 1    root.findWordCount = 1

        root.children = null      root.children = null

     \*/

    private int helper(String s, TrieNode root, int curPos) {

        if(root == null) {

            return 0;

        }

        // Reverse logic here, not like normal find char from 's' in Trie tree,

        // instead, we try to find root's char back in 's', because basically

        // we are trying to map each word (each like a top to bottom node path)

        // inside Trie tree back to 's', so we are recursively searching current

        // root's char's first index beyond(inclusive as >=) current position in

        // 's' (since DFS should only look for the char on forward indexes rather

        // than backward indexes which already checked in previous recursions),

        // and if we find any node has 'findWordCount' not as 0 on searching path,

        // which means find a word (as a node path on Trie tree) as subsequence

        // in 's', we should add into final count, the covergence DFS logic similar

        // to L1034.Coloring A Border.

        // In other words, the DFS method try to walk through each word (equal to each

        // path) in Trie tree and project word's each character back to 's', looking for

        // each character's first potential index in 's' but on forwarding indexes

        // only, to identify this first potential index, we have to leverage 'indexOf'

        // method and also each recursion level need to record current position in 's',

        // then in next recursion level only search beyond current position in 's'

        // to make sure not return old index of this character if any, the final

        // effect will be like run each word many times against 's'.

        // Note: Don't try to use binary search against 's' for first index of the

        // char(root.c), since its not a sorted object.

        int firstIndexOfChar = s.indexOf(root.c, curPos);

        // If the character is not found (first it should not be the dummy root char

        // as "\*", but a real char from word, if a dummy root char, we should not

        // return directly but continue search), return 0

        if(root.c != '\*' && firstIndexOfChar == -1) {

            return 0;

        }

        int count = root.findWordCount;

        for(int i = 0; i < 26; i++) {

            // If current node is root(because its character dummy as '\*'), the first

            // index of character ('root.chidren[i].c') should perform DFS against 's'

            // start from index 0, otherwise should only beyond 'firstIndexOfChar'

            count += helper(s, root.children[i], root.c == '\*' ? 0 : firstIndexOfChar + 1);

        }

        return count;

    }

    private TrieNode createTrieTree(String[] words) {

        TrieNode root = new TrieNode('\*');

        for(String word : words) {

            addWord(word, root);

        }

        return root;

    }

    private void addWord(String word, TrieNode root) {

        for(char c : word.toCharArray()) {

            int index = c - 'a';

            if(root.children[index] == null) {

                root.children[index] = new TrieNode(c);

            }

            root = root.children[index];

        }

        root.findWordCount++;

    }

}

Time Complexity: O((N × M) × |s|)

Space Complexity: O(N × M)

N be the total number of words

M be the average length of the words

performing indexOf across s costs O(|s|)

**Refer to**

<https://leetcode.com/problems/number-of-matching-subsequences/solutions/157065/java-trie-solution/>

class Solution {

    // Definition of TrieNode

    class TrieNode {

        char c; // Character stored in this node

        int end; // Count of words that end at this node

        TrieNode[] children; // References to child nodes (for each character 'a'-'z')

        public TrieNode(char c) {

            this.c = c;

            this.end = 0;

            this.children = new TrieNode[26]; // Initialize with 26 children for each alphabet

        }

    }

    public int numMatchingSubseq(String S, String[] words) {

        if (words == null || words.length == 0) return 0;

        // Build the Trie with the given words

        TrieNode root = createTrie(words);

        // Use Depth-First Search (DFS) to count matching subsequences in S

        return dfs(S, root, 0);

    }

    // Build a Trie from the list of words

    private TrieNode createTrie(String[] words) {

        TrieNode root = new TrieNode('\*'); // Root node initialized with a special character '\*'

        for (String word : words) {

            addWord(root, word);

        }

        return root;

    }

    // Helper method to add a word to the Trie

    private void addWord(TrieNode root, String word) {

        for (char c : word.toCharArray()) {

            int index = c - 'a'; // Calculate the index for the character

            if (root.children[index] == null) {

                root.children[index] = new TrieNode(c);

            }

            root = root.children[index]; // Move to the child node

        }

        root.end++; // Increment the count of words that end at this node

    }

    // DFS to find the number of matching subsequences

    private int dfs(String S, TrieNode root, int pos) {

        if (root == null) return 0;

        // Find the first occurrence of the current node's character in S, starting from position `pos`

        int index = S.indexOf(root.c, pos);

        // If the character is not found (and it's not the root), return 0

        if (root.c != '\*' && index == -1) return 0;

        // Add the count of words that end at this node

        int res = root.end;

        // Recursively explore all children of the current node

        for (int i = 0; i < 26; i++) {

            res += dfs(S, root.children[i], root.c == '\*' ? 0 : index + 1);

        }

        return res;

    }

}

**Refer to chatGPT**

**Key Insights:**

Typical Usage of a Trie:

In most cases, we construct a Trie from a collection of strings and then **search for a specific string or prefix** within the Trie.

This leverages the Trie's structural advantage of prefix matching, allowing efficient queries for strings or prefixes.

Usage of Trie in L792:

Here, a Trie is constructed from the given words array, but instead of searching directly for a string in the Trie, we use it as a **storage and traversal mechanism**.

The DFS extracts each word from the Trie one character at a time and **projects it back** onto the input string s by searching for matching characters in sequence.

The goal is not to search for words within the Trie itself but to check if the words stored in the Trie exist as subsequences in s.

Why This Feels Atypical:

The solution doesn't leverage the full capability of a Trie (like prefix searching or direct lookups).

Instead, it uses the Trie's **hierarchical structure** to organize words efficiently and the findWordCount at each node to count matching words during DFS traversal.

The Trie here acts as a compact storage mechanism to group and organize words for efficient traversal rather than as a direct search tool.

Storage Advantage:

The Trie helps consolidate common prefixes of words, reducing redundant comparisons (e.g., if two words share a prefix, their characters are processed together).

This minimizes the overhead of processing duplicate prefixes in the words array during the projection back onto s.

**Time Complexity:**

**1. Building the Trie**

Insertion of each word: For a word of length L, inserting it into the Trie takes O(L), as each character is added sequentially.

Total insertion for all words: Let **N be the total number of words** in the words array, and let **M be the average length of the words**.The total time for building the Trie is:
O(Total characters in all words)=O(N×M)

**2. Matching Subsequences Using DFS**

DFS traversal for each word path in the Trie:

For each TrieNode, we find the first occurrence of its character in the string s starting at position curPos using indexOf.

The cost of finding a character using indexOf is proportional to the remaining length of s being searched, which can be up to O(|s|) in the worst case.

Total DFS calls: In the worst case, all nodes in the Trie will be visited. Let K be the total number of nodes in the Trie. K is proportional to the sum of the lengths of all words, i.e., K≈N×M.

Worst-case DFS cost: For each TrieNode, **performing indexOf across s costs O(|s|)**. Thus, the total cost of the DFS step is:
O(K×|s|)=O((N×M)×|s|)

**Total Time Complexity**

Combining both steps:

Building the Trie: O(N×M)

DFS matching: O((N×M)×|s|)

Final time complexity: O((N×M)×|s|)

**Space Complexity:**

**1. Space for the Trie**

Each node in the Trie stores:

A character (char c) → constant space.

A count for the end of words (int findWordCount) → constant space.

An array of children (TrieNode[] children) → O(26) for each node, though most of the entries will be null.

Total number of nodes in the Trie:

In the worst case, there will be a separate node for each character in every word in the words array.

Let N be the number of words and M be the average length of the words. The total number of nodes in the Trie is proportional to the total number of characters across all words, i.e., N×M.

Space for the Trie: Each node occupies O(26)=O(1)), so the space for the Trie is:O(N×M)

**2. Space for the Recursive Call Stack**

The depth of the Trie is at most equal to the length of the longest word in the words array, i.e., maxWordLength.

In the worst case, the DFS will recurse to the maximum depth of the Trie. Hence, the maximum stack space used is proportional to maxWordLength.

Space for the DFS call stack:O(maxWordLength)

**3. Space for Auxiliary Structures**

Input string **s and words array**: These are given inputs and do not contribute to additional space usage.

Temporary variables: The variables used for each DFS call (like curPos, firstIndexOfChar, etc.) occupy constant space.

**Total Space Complexity**

Combining all the above:

Trie structure: O(N×M)

Recursive stack: O(maxWordLength)

Final space complexity:O(N×M) (if N×M > maxWordLengthN, otherwise the stack size may dominate in specific edge cases).

**Solution 3: Binary Search (60 min, exactly same as** [L392.Is Subsequence (Ref.L792)](note://WEBc024c0d4680935a41132791e54614e47)**, TLE 23/53)**

class Solution {

    public int numMatchingSubseq(String s, String[] words) {

        int count = 0;

        for(String word : words) {

            count += isSubsequence(word, s);

        }

        return count;

    }

    private int isSubsequence(String word, String s) {

        // {k, v} -> {char in s, indexes of this char in s}

        Map<Character, List<Integer>> s\_char\_indexes = new HashMap<>();

        for(int i = 0; i < s.length(); i++) {

            s\_char\_indexes.putIfAbsent(s.charAt(i), new ArrayList<>());

            s\_char\_indexes.get(s.charAt(i)).add(i);

        }

        int prev\_char\_last\_index = -1;

        for(int i = 0; i < word.length(); i++) {

            List<Integer> indexes = s\_char\_indexes.get(word.charAt(i));

            // Must check if current char exist in 's'

            // Test out: word = "axc", s = "ahbgdc"

            if(indexes == null) {

                return 0;

            }

            // Use binary search to find current char's smallest new index

            // which should greater than previous char's last index

            int curr\_char\_new\_index = binarySearch(indexes, prev\_char\_last\_index);

            // If no valid index is found for current char, 'word' is not

            // a subsequence for 's'

            if(curr\_char\_new\_index == -1) {

                return 0;

            }

            // Update previous char's last index to the found index

            // of current char's smallest new index

            prev\_char\_last\_index = curr\_char\_new\_index;

        }

        return 1;

    }

    // Find lower boundary

    private int binarySearch(List<Integer> indexes, int prev\_char\_last\_index) {

        int lo = 0;

        int hi = indexes.size() - 1;

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

            // Strictly '>' since bottom line is new index one more

            // larger than given index

            if(indexes.get(mid) > prev\_char\_last\_index) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        // If 'lo' is within bounds, return the index, otherwise, return -1

        if(lo >= indexes.size()) {

            return -1;

        }

        return indexes.get(lo);

    }

}

Time Complexity: O(N \* S + N \* W \* log S)

In this isSubsequence method, they create a Map<Character, List<Integer>> called s\_char\_indexes

for each word. This means that for every single word, they are reprocessing the entire string s

to build this map of character indices. So, if there are N words, this preprocessing step is

done N times, which is O(N \* S) where S is the length of s. This is very inefficient compared

to the binary search solution, which preprocesses s once in O(S) time.

Space Complexity: O(N)

**The above Binary Serach solution encounters TLE and below is analysis**

**Refer to Deepseek**

Example Binary Search solution which won't encounters TLE and pass all 53 test cases in 140 ms

class Solution {

public int numMatchingSubseq(String s, String[] words) {

List<Integer>[] charIndices = new ArrayList[26];

for (int i = 0; i < 26; i++) {

charIndices[i] = new ArrayList<>();

}

for (int i = 0; i < s.length(); i++) {

char c = s.charAt(i);

charIndices[c - 'a'].add(i);

}

int count = 0;

for (String word : words) {

if (isSubsequence(word, charIndices)) {

count++;

}

}

return count;

}

private boolean isSubsequence(String word, List<Integer>[] charIndices) {

int currentPos = -1;

for (char c : word.toCharArray()) {

List<Integer> indices = charIndices[c - 'a'];

if (indices.isEmpty()) {

return false;

}

int key = currentPos + 1;

int bsResult = Collections.binarySearch(indices, key);

if (bsResult >= 0) {

currentPos = indices.get(bsResult);

} else {

int insertionPoint = - (bsResult + 1);

if (insertionPoint >= indices.size()) {

return false;

}

currentPos = indices.get(insertionPoint);

}

}

return true;

}

}

Okay, let's try to figure out why the user's solution is getting a Time Limit Exceeded (TLE) while the binary search solution passes all test cases. Both solutions use binary search to determine if a word is a subsequence of the string s, but there must be some key differences in their implementations that affect performance.

First, let's look at the binary search solution that works. In this solution, they preprocess the string s by creating an array of lists, where each list contains the indices of a particular character in s. For example, charIndices[0] would have all the indices where 'a' appears in s. This preprocessing is done once, outside the loop that checks each word. Then, for each word, they check if it's a subsequence by using binary search on these precomputed lists. This way, the preprocessing happens once, and each word is checked efficiently using the stored indices.

**Now, let's examine the user's solution that's getting TLE. The user's approach also uses binary search, but there's a critical difference. In the user's isSubsequence method, they create a Map<Character, List<Integer>> called s\_char\_indexes for each word. This means that for every single word, they are reprocessing the entire string s to build this map of character indices. So, if there are N words, this preprocessing step is done N times, which is O(N \* S) where S is the length of s. This is very inefficient compared to the binary search solution, which preprocesses s once in O(S) time.**

Another thing to note is that in the user's solution, the binarySearch method is implemented manually. While the logic seems correct, using the built-in Collections.binarySearch might be more optimized in practice. The user's binary search could have slight inefficiencies, but the main issue is definitely the repeated preprocessing of s for each word.

So, the key problem in the user's solution is the repeated creation of the character indices map for every word. This leads to a much higher time complexity, especially when the number of words is large. The binary search solution avoids this by preprocessing once, making it significantly faster for large inputs.

To fix the user's solution, they should preprocess the character indices once before processing all words, similar to the binary search solution. By moving the creation of s\_char\_indexes outside of the isSubsequence method and doing it once at the beginning, the time complexity would improve from O(N \* S + N \* W \* log S) to O(S + N \* W \* log S), where W is the average length of the words. This would prevent redundant computations and eliminate the TLE issue.

The key difference lies in how the character indices of s are preprocessed. Here's the breakdown:

**1. Preprocessing Overhead**

Your Solution:

For every word, you build the s\_char\_indexes map from scratch. This involves iterating through s and populating the map with character indices each time.

Time Complexity: For N words, this becomes O(N·S), where S is the length of s.

Binary Search Solution:

Preprocesses s once into charIndices, an array of lists storing indices for each character.

Time Complexity: O(S) (done once), leading to O(S + N·W·logS), where W is the average word length.

**2. Why Your Solution Gets TLE**

Repeatedly rebuilding s\_char\_indexes for every word is redundant and computationally expensive. For example, if s has 1e4 characters and there are 1e4 words, your solution does 1e8 operations just for preprocessing, which is impractical.

**3. Fixing Your Solution**

Move the preprocessing of s outside the word loop to avoid redundant computations:

class Solution {

    public int numMatchingSubseq(String s, String[] words) {

        // Preprocess 's' only once

        // {k, v} -> {char in s, indexes of this char in s}

        Map<Character, List<Integer>> sCharIndices = new HashMap<>();

        for (int i = 0; i < s.length(); i++) {

            char c = s.charAt(i);

            sCharIndices.putIfAbsent(c, new ArrayList<>());

            sCharIndices.get(c).add(i);

        }

        int count = 0;

        for (String word : words) {

            count += isSubsequence(word, sCharIndices);

        }

        return count;

    }

    private int isSubsequence(String word, Map<Character, List<Integer>> sCharIndices) {

        int prev\_char\_last\_index = -1;

        for (char c : word.toCharArray()) {

            List<Integer> indices = sCharIndices.get(c);

            // Must check if current char exist in 's'

            // Test out: word = "axc", s = "ahbgdc"

            if (indices == null) {

                return 0;

            }

            // Use binary search to find current char's smallest new index

            // which should greater than previous char's last index

            int curr\_char\_new\_index = binarySearch(indices, prev\_char\_last\_index);

            // If no valid index is found for current char, 'word' is not

            // a subsequence for 's'

            if (curr\_char\_new\_index == -1) {

                return 0;

            }

            // Update previous char's last index to the found index

            // of current char's smallest new index

            prev\_char\_last\_index = curr\_char\_new\_index;

        }

        return 1;

    }

    // Find lower boundary

    private int binarySearch(List<Integer> indexes, int prev\_char\_last\_index) {

        int lo = 0;

        int hi = indexes.size() - 1;

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

            // Strictly '>' since bottom line is new index one more

            // larger than given index

            if(indexes.get(mid) > prev\_char\_last\_index) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        // If 'lo' is within bounds, return the index, otherwise, return -1

        if(lo >= indexes.size()) {

            return -1;

        }

        return indexes.get(lo);

    }

}

Time Complexity: O(S + N·W·logS)

Space Complexity: O(N)

**Key Takeaways**

Preprocess Once: Avoid rebuilding the same data structure multiple times.

Leverage Built-in Methods: Use Collections.binarySearch (as in the original solution) for better optimization.

Time Complexity: Your fixed solution will now run in O(S + N·W·logS), matching the efficiency of the binary search approach.

**Solution 4: Two Pointers (10 min, exactly same as** [L392.Is Subsequence (Ref.L792)](note://WEBc024c0d4680935a41132791e54614e47)**, TLE 45/53)**

class Solution {

    public int numMatchingSubseq(String s, String[] words) {

        int count = 0;

        for(String word : words) {

            count += isSubsequence(word, s);

        }

        return count;

    }

    private int isSubsequence(String word, String s) {

        int i = 0;

        int j = 0;

        while(j < s.length()) {

            if(word.charAt(i) == s.charAt(j)) {

                i++;

                if(i == word.length()) {

                    return 1;

                }

            }

            j++;

        }

        return 0;

    }

}

Time Complexity: O(n^2)

Space Complexity: O(1)

**Refer to**

[L208.Implement Trie (Prefix Tree)](note://68E95CFADD3A4906B6290D7096A7EFAD)

[L392.Is Subsequence (Ref.L792)](note://WEBc024c0d4680935a41132791e54614e47)

[L1034.Coloring A Border (Ref.L200,L733)](note://WEB1a530db8c1e1bd0e15c9dff995bd2b72)

[L2062.Count Vowel Substrings of a String (Ref.L2461)](note://WEBffe1facb0ba61c1f4e6681e23ad781cb)