792. Number of Matching Subsequences Medium String ) Hash Table **Binary Search** Array **Dynamic Programming** 

string is defined as a sequence that can be derived from the original string by removing some characters without altering the order of the remaining characters. For instance, "ace" is a subsequence of "abcde" because you can remove 'b' and 'd' to get "ace". Your goal is to return the count of strings in words that can be formed in this way from the string s.

The problem requires you to determine how many strings in the array words are subsequences of the string s. A subsequence of a

Sorting

**Leetcode Link** 

matched the word in words [i] with characters in s.

**Problem Description** 

Intuition To solve this problem, the key idea is to track the progress of each word in the words array as we iterate through the string s. Instead of checking each word against the entire string s at once, which would be inefficient, we use a smarter approach. We create a dictionary that maps the first character of each word to a queue holding tuples that represent the words. Each tuple consists of an

index i and a position j, where i is the index of the word in the words array, and j is the position up to which we have successfully

queue, if we find a matching character in s, we move to the next character in the word by incrementing j. If we reach the end of a word, it means we have successfully found it as a subsequence in s, and we increment our answer count (ans). We use a queue because it efficiently supports removing elements from the front, allowing us to process the words in the order they are expecting to match characters from s. By organizing the words based on the next expected character, we only consider the

As we iterate through the string s, we get the queue of words (tuples) that are expecting the current character. For each word in the

**Solution Approach** The solution approach makes use of the following algorithms, data structures, and patterns:

• Dictionary with Queue: A defaultdict(deque) from Python's collections module is used to map the first character of each

word in words to a queue containing tuples of the form (i, j). Here i is the index of the word in the array words, and j is the

# • Character Iteration and Matching: The solution iterates through each character c in the string s and uses it to reference the queue of tuples waiting for that character (d[c]).

• Queue Processing: Within this iteration, another loop runs for the length of the queue d[c], which processes each tuple. For

character is found, and recounts every time a word is completely matched as a subsequence.

words = ["a", "bb", "acd", "ace"]. We want to find how many words are subsequences of s.

relevant part of words for each character in s, making the process efficient.

current character position we have matched in the word so far.

- each tuple (i, j) dequeued from d[c], the j is incremented to point to the next character since the current one c is a match. • Subsequence Completion Check: If the incremented j equals the length of the word words [i], it means all the characters in words[i] are matched in sequence with some characters of s, therefore words[i] is a subsequence of s. When this condition is
- met, the answer counter ans is incremented. • Updating the Dictionary with Queues: If j is not equal to the length of the word, the tuple (i, j) is not completed yet and must

wait for the next character words[i][j]. Hence, it is appended back into the queue corresponding to this next expected

character d[words[i][j]]. In essence, the algorithm processes s character by character, moving 'pointers' through each word in words when a matching

The algorithm complexity is mainly O(N + M), where N is the length of the string s, and M is the total length of all words in words. It's

because each character of each word is processed at most once when the corresponding character in s is found. **Example Walkthrough** Let's walk through a small example to illustrate the solution approach in action. Consider the string s = "abcde" and the list of words

and the initial character position 0. 'a': deque([(0, 0), (2, 0), (3, 0)]), 'b': deque([(1, 0)]),

To begin, we initialize our dictionary with queues, d, that will hold the progress for each word in words as we iterate over s. For our

example, the initial state of the dictionary will be as follows since each queue will contain tuples with the index of the word in words

## 1. For the first character 'a' in s, we look in d to find the queue for 'a', which is deque([(0, 0), (2, 0), (3, 0)]). We have three words waiting to match the character 'a' in s.

'b': deque([(1, 0)]),

And the dictionary updates to:

'c': deque([(2, 1), (3, 1)]),

enqueue tuple (2, 2) since we are now waiting for 'd'.

4. The next character in s is 'd', which affects the queue under 'd'.

'c': deque([(2, 1), (3, 1)]),

Now we start iterating over s, character by character:

the tuple since "acd" is not yet fully matched. Now the tuple is (2, 1).

Now our dictionary looks like this:

a. dequeue (0, 0): Since 'a' in s matches the first character of words [0] which is "a", it's a complete match. We increment ans to 1.

b. dequeue (2, 0): Now we're looking at words [2] which is "acd". The character 'a' matched, so we increment j to 1 and requeue

c. dequeue (3, 0): Similarly, for words [3] which is "ace", we have a match on 'a', increment j to 1, and requeue the tuple to (3, 1).

a. dequeue (1, 0): We look at words [1] which is "bb". Since 'b' matches the first character, we increment j to 1 and enqueue the tuple (1, 1) back because it's still awaiting another 'b'.

2. The second character of s is 'b'. We check our dictionary for the queue under 'b', which is deque([(1, 0)]).

'b': deque([(1, 1)]),

a. dequeue (2, 1): "acd" in words[2] has matched 'a' and now 'c', so j becomes 2. Since j is not equal to the length of "acd", we

b. dequeue (3, 1): Similarly, "ace" in words[3] has matched 'a' and 'c', and j is incremented to 2, indicating we are waiting for 'e'

a. dequeue (2, 2): The tuple for "acd" is fully matched now because 'd' was the last needed character, so we increment ans by 1

a. dequeue (3, 2): Finally, "ace" is fully matched, and our total count ans is incremented by one more (total ans is now 3).

because we've found that three out of four words in words are subsequences of s. The word "bb" was not a subsequence since the

The end state of the dictionary is not relevant anymore since we have processed all of s. The algorithm finishes with ans = 3

'd': deque([(2, 2)]), 'e': deque([(3, 2)]),

Dictionary after the operation:

'b': deque([(1, 1)]),

'e': deque([(3, 2)]),

second 'b' couldn't be matched.

from collections import defaultdict, deque

char\_to\_word\_indices = defaultdict(deque)

for index, word in enumerate(words):

matching\_count = 0

return matching\_count

from typing import List

**Java Solution** 

class Solution {

Python Solution

class Solution:

14

17

18

19

20

22

23

24

38

39

40

41

42

43

10

11

12

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

35

36

37

38

39

40

C++ Solution

class Solution {

2 public:

Args:

'b': deque([(1, 1)]),

Updated dictionary:

(total ans is now 2).

next.

```
5. The last character in s is 'e', and we process the queue under 'e'.
```

def num\_matching\_subseq(self, string: str, words: List[str]) -> int:

3. Next, s gives us 'c'. We process the queue under 'c', which is deque([(2, 1), (3, 1)]).

string (str): the string to be searched for subsequences. words (List[str]): the list of words to check as potential subsequences. 10 11 12 Returns: int: the number of words that are subsequences of 'string'. 13

Count the number of words from the list 'words' that are subsequences of the string 'string'.

# Each tuple contains an index of a word in words and the next character index to search for.

# Initialize a dictionary to hold queues of tuples for each character in words.

# Populate the dictionary with initial character indices for each word.

next\_char = words[word\_index][char\_index]

Remember to include the necessary import statement at the top for List from typing:

// Create an array of dequeues to store positions of characters in words

// Fill the deques with the first characters of each word in words array

waitingChars[words[i].charAt(0) - 'a'].offer(new int[]{i, 0});

int matches = 0; // Store the number of matching subsequences

// Get the deque corresponding to the current character

char\_to\_word\_indices[word[0]].append((index, 0))

# Counter for the number of matching subsequences found.

# Loop through each character in the input string.

# Return the total count of matching subsequences.

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

Arrays.setAll(waitingChars, k -> new ArrayDeque<>());

// Iterate through each character of the string s

Deque<int[]> queue = waitingChars[c - 'a'];

for (int i = queue.size(); i > 0; --i) {

int[] pair = queue.pollFirst();

// Dequeue the first element

Deque<int[]>[] waitingChars = new Deque[26];

// Initialize each dequeue in the array

for (int i = 0; i < words.length; ++i) {</pre>

// Process all elements in queue

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

return matches;

for char in string: 29 # Process all tuples (word index, character index) for the current character. for \_ in range(len(char\_to\_word\_indices[char])): 30 word\_index, char\_index = char\_to\_word\_indices[char].popleft() 31 32 char\_index += 1 # Move to the next character in the current word 33 # If we reached the end of the word, increment the count of matching subsequences. 34 35 if char\_index == len(words[word\_index]): 36 matching\_count += 1 else: 37

# Else, append the tuple (word index, next character index) to the appropriate list.

char\_to\_word\_indices[next\_char].append((word\_index, char\_index))

++matches; 29 } else { 30 // Otherwise, enqueue the pair back with the updated character index 31 waitingChars[words[wordIndex].charAt(charIndex) - 'a'].offer(new int[]{wordIndex, charIndex}); 33 34

int wordIndex = pair[0], charIndex = pair[1] + 1;

if (charIndex == words[wordIndex].length()) {

// Return the total number of matching subsequences found

int numMatchingSubseq(string s, vector<string>& words) {

vector<queue<pair<int, int>>> charQueues(26);

// Create a vector of queues to store pairs of indices.

function numMatchingSubseq(s: string, words: string[]): number {

const charQueues: Array<Queue<[number, number]>> = Array.from(

const charIndex = words[i].charCodeAt(0) - 'a'.charCodeAt(0);

// Reference to the queue corresponding to the current character.

// Get the front pair (word index, char index) and remove it from the queue.

// If all characters in the word have been found in `s`, increment the answer.

// Otherwise, push the pair (word index, next char index) to the corresponding queue

const nextCharCode = words[wordIndex].charCodeAt(nextCharIndex) - 'a'.charCodeAt(0);

// Increment the char index to check the next character of the word.

// for the next character to be checked in `words[wordIndex]`.

charQueues[nextCharCode].enqueue([wordIndex, nextCharIndex]);

const queue = charQueues[c.charCodeAt(0) - 'a'.charCodeAt(0)];

const [wordIndex, charIndex] = queue.dequeue();

if (nextCharIndex === words[wordIndex].length) {

// Create an array of queues to store pairs of indices.

() => new Queue<[number, number]>());

charQueues[charIndex].enqueue([i, 0]);

// Iterate over each character in the string `s`.

const nextCharIndex = charIndex + 1;

// Process each pair in the queue.

for (let t = 0; t < size; ++t) {

const size = queue.size();

ans++;

} else {

for (let i = 0; i < words.length; ++i) {

// If the whole word is found, increment the match count

```
// Populate the queues with pairs: the index of the word in the `words` vector and the character index being checked.
9
           for (int i = 0; i < words.size(); ++i) {</pre>
10
                charQueues[words[i][0] - 'a'].emplace(i, 0);
11
12
13
14
           // Initialize the answer to 0.
15
           int ans = 0;
16
17
           // Iterate over each character in the string `s`
           for (char& c : s) {
18
               // Reference to the queue corresponding to the current character
19
                auto& queue = charQueues[c - 'a'];
20
21
22
               // Process each pair in the queue
23
                for (int t = queue.size(); t > 0; --t) {
                   // Get the front pair (word index, char index) and remove it from the queue
24
                    auto [wordIndex, charIndex] = queue.front();
25
26
                    queue.pop();
27
28
                   // Increment the char index to check the next character of the word
29
                    ++charIndex;
30
31
                   // If all characters in the word have been found in `s`, increment the answer
32
                   if (charIndex == words[wordIndex].size()) {
33
                        ++ans;
34
                    } else {
35
                        // Otherwise, push the pair (word index, next char index) to the corresponding queue
                        // for the next character to be checked in `words[wordIndex]`
36
                        charQueues[words[wordIndex][charIndex] - 'a'].emplace(wordIndex, charIndex);
37
38
39
40
           // Return the total count of matching subsequences
41
42
           return ans;
43
44 };
45
```

1 // Define a method to count the number of words from the `words` array that are subsequences of the string `s`.

// Each element represents a queue for words starting with the corresponding character ('a' to 'z').

// Populate the queues with pairs: the index of the word in the `words` array and the character index being checked.

// Function to count the number of words from the `words` vector that are subsequences of the string `s

// Each element represents a queue for words starting with the corresponding character ('a' to 'z').

## 41 42 43 44 // Return the total count of matching subsequences. 45 return ans; 46

class Queue<T> {

// Helper class for Queue

dequeue(): T {

size(): number {

Time Complexity

private data: T[] = [];

enqueue(item: T): void {

this.data.push(item);

return this.data.shift();

return this.data.length;

Time and Space Complexity

the number of words in words.

Typescript Solution

let ans = 0;

for (const c of s) {

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

47

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

{ length: 26 },

// Initialize the answer to 0.

```
characters in words list, but initializing the deques themselves takes O(1) time.
• The first loop goes through all the words in words list to populate the dictionary d. Each word is processed in O(1) time to append
 its index and starting character position to the corresponding deque, resulting in O(n) time complexity for this step, where n is
```

The time complexity of the given code can be analyzed as follows:

Since each word is exactly processed once, and each character in s results in at most one deque operation (either popleft or append), the number of operations is proportional to the length of s plus the total length of all words. Hence, the time complexity, in this case, is O(m + k), where m is the length of s and k is the sum of lengths of all words in words.

**Space Complexity** The space complexity of the code is determined by:

• Initializing the dictionary d with a deque for each unique character has a space complexity that depends on the number of unique

• The second loop goes through each character in the input string s. For each character c, it processes all indices (i, j) in d[c].

• The dictionary d which contains as many keys as there are unique starting characters in the words, and each key has a deque that contains at most n tuples (i, j), where n is the number of words. However, since each word can only be in one deque at a time,

Thus, the overall time complexity combines the complexities of both loops, resulting in O(n + m + k).

 There are no other data structures that require significant space. Therefore, the space complexity is O(n), where n is the number of words in words.

it's more precise to say that it contains a total of n tuples across all deques.