# 524. Longest Word in Dictionary through Deleting

Sorting

String

**Problem Description** 

**Array** 

**Two Pointers** 

Medium

characters from a given string s. If you can make multiple words of the same maximum length, you should return the one that comes first alphabetically. Should there be no words from the dictionary that can be formed, the answer would be an empty string. To solve this problem, we need to determine if a given word in the dictionary can be formed by deleting characters from the

The task is to find the longest string from a given list of words (the dictionary) that can be created by removing some of the

string s. We use a two-pointer approach to compare characters of a dictionary word and the string s without rearranging any character's order.

#### The intuition behind the solution is to use a two-pointer approach that can efficiently validate whether a word from the dictionary is a subsequence of the string s. Here's how we proceed:

Intuition

• Initialize two pointers, i for the index in the string s and j for the index in the current dictionary word. • Increment i each time we examine a character in s.

• Increment j only when the characters at i in s and j in the dictionary word match.

- A dictionary word is a subsequence of s if we can traverse through the entire word (i.e., j equals the length of the word) by selectively matching
- characters in s.
- formed. During the process, we also keep track of the longest word that satisfies the condition. If multiple words have the same length, we choose the one with the lowest lexicographical order, which is the same as saying the smallest in alphabetical order.

The check function implements the two-pointer technique, and the outer loop through the dictionary selects the best candidate

By iterating through each word in the dictionary and applying the two-pointer technique, we can check which words can be

word according to the above-mentioned criteria. **Solution Approach** 

The solution implements a straightforward algorithm which utilizes the two-pointer pattern to match dictionary words against the string s. Let's break down how the Solution class achieves this:

The findLongestWord function is where we start, and it takes a string s and a list of strings dictionary as inputs.

# It defines an inner function check that takes two strings a (the given string s) and b (a word from the dictionary). This function

uses the two-pointer technique to determine if b can be formed from a by deletion of characters. It starts with two indexes, i at 0 for string a and j at 0 for string b. It iterates over the characters in a using i and only moves j forward if the characters at a[i] and b[j] match.

- ∘ If j reaches the length of the string b, it means b is a subsequence of a, and check returns True. After the check function, we have a variable ans which is initialized to an empty string. This will hold the longest string from
- the dictionary that can be formed. The solution iterates over each word in the dictionary:

If it can, it then checks if the current word is longer than the one stored in ans or if it is the same length but lexicographically smaller.

• If either condition is met, we update ans with the current word. After checking all words in the dictionary, the solution returns ans, which contains the longest word that can be formed by

Using the check function, it verifies if the current word can be formed from s.

The use of the two-pointer technique is a key aspect of this solution as it allows for an efficient check without the need to create

deleting some of the given string characters, or the smallest one in lexicographical order in case there are multiple.

additional data structures or perform unnecessary computations. It is a common pattern when you need to compare or match

• s = "abpcplea" dictionary = ["ale", "apple", "monkey", "plea"] The goal is to find the longest word from dictionary that can be formed by deleting some characters in s.

### For each word in the dictionary: Call the check function to determine if the word can be formed from s.

from s.

**Python** 

According to our algorithm:

**Example Walkthrough** 

 Traverse s from left to right using i and compare with j on "ale" Characters match at s[0] = "a" and "ale"[0] = "a", so increment both i and j.

Start by iterating over each word in the dictionary. Initialize ans = "".

Let's consider a small example to illustrate the solution approach. Assume we have the following:

 Update ans with "ale" as it is currently the longest word found. Next, for the word "apple":

• Initialize two pointers, i = 0 for s and j = 0 for "ale".

Let's start with the word "ale":

sequences without altering their order.

• Repeat the same check procedure. • The word "apple" is also found to be a subsequence within s by matching "a", "p", "p", "l" and skipping the unused characters.

Since "apple" is longer than "ale", update ans with "apple".

Then, check the word "monkey": It is found that not all characters can be matched; thus, it is not a subsequence of s. No need to update ans. Finally, for the word "plea":

Since "plea" is the same length as "apple", but not lexicographically smaller, we do not update ans.

also adheres to the lexicographical order in case of length ties (even though there were none in this case).

• The check function will confirm that "plea" is a subsequence of s.

Since s [1] is "b" and doesn't match "ale" [1] ("I"), only i is incremented.

When j reaches the end of "ale", we know it is a subsequence of s.

Continue incrementing i until we find a match for each character of "ale".

Solution Implementation

# Traverse both strings and check if b is subsequence of a

pos\_b += 1 # Move pointer of string b if characters match

# Update longest\_word if word is longer or lexicographically smaller

// Function to find the longest word in the dictionary that can be formed by deleting

// Check if current word can be formed by deleting some characters from s

public String findLongestWord(String s, List<String> dictionary) {

if (longestWord.length() < word.length() ||</pre>

// Helper method to check if string a is a subsequence of string b

if is\_subsequence(s, word) and (len(longest\_word) < len(word) or</pre>

m, n = len(a), len(b)

while pos\_a < m and pos\_b < n:</pre>

longest\_word = word

// some characters of the given string s.

for (String word : dictionary) {

if (isSubsequence(s, word)) {

longestWord = word;

private boolean isSubsequence(String a, String b) {

**if** (a[i] == b[j]) ++j;

dictionary.sort((word1, word2) => {

if (word1.length === word2.length) {

return word2.length - word1.length;

// Store the length of the string `s`.

// Iterate over the sorted dictionary.

for (const targetWord of dictionary) {

const stringLength = s.length;

continue;

let stringPointer = 0;

let targetPointer = 0;

return word1.localeCompare(word2);

// Store the length of the current target word.

// Iterate over the characters in `s` and the target word.

if (s[stringPointer] === targetWord[targetPointer]) {

// then the target word can be formed. Return it as the answer.

const targetLength = targetWord.length;

if (targetLength > stringLength) {

targetPointer++;

if (targetPointer === targetLength) {

return targetWord;

return j == bLength;

**}**;

**TypeScript** 

});

++i; // Always increment i to move forward in string a

// String b is a subsequence of a if j has reached the end of b

// If two words have the same length, sort them lexicographically in ascending order.

// If the target word is longer than the string `s`, it cannot be formed.

while (stringPointer < stringLength && targetPointer < targetLength) {</pre>

// If all characters of the target word have been found in `s` in order,

// Initialize two pointers for comparing characters in `s` and the target word.

// If the current characters match, move the target pointer to the next character.

function findLongestWord(s: string, dictionary: string[]): string {

// Sort the dictionary in descending order by word length.

String longestWord = "";

return longestWord;

**if** a[pos\_a] == b[pos\_b]:

# Iterate over each word in the given dictionary

# Check if the word is a subsequence of s

pos a = pos b = 0

longest\_word = ''

for word in dictionary:

class Solution: def findLongestWord(self, s: str, dictionary: List[str]) -> str: # Helper function to check if b is a subsequence of a def is\_subsequence(a, b):

After checking all words, ans contains "apple", which is the longest word that can be formed by deleting some characters

The result from our example is "apple", as it is the longest word that can be created from s by deleting some characters, and it

pos\_a += 1 # Always move pointer of string a # Check if reached the end of string b, meaning b is a subsequence of a return pos\_b == n # Initialize the answer to an empty string

// Update longestWord if current word is longer, or the same length but lexicographically smaller

(longestWord.length() == word.length() && word.compareTo(longestWord) < 0)) {</pre>

(len(longest\_word) == len(word) and longest\_word > word)):

```
# Return the longest word that is a subsequence of s and satisfies the condition
       return longest_word
Java
```

class Solution {

```
int i = 0; // Pointer for string a
        int j = 0; // Pointer for string b
        int m = a.length();
        int n = b.length();
       while (i < m && j < n) {</pre>
            if (a.charAt(i) == b.charAt(j)) {
                // If current characters match, move pointer j to next position in string b
                ++j;
            // Always move pointer i to next position in string a
            ++i;
        // If we have traversed the entire string b, it means it is a subsequence of a
        return j == n;
C++
class Solution {
public:
    // Function to find the longest string in the dictionary that is a
    // subsequence of s. If there are multiple, the smallest lexicographically
    // will be returned.
    string findLongestWord(string s, vector<string>& dictionary) {
        string longestWord = ""; // Initialize the longest word to an empty string
        // Iterate over each word in the dictionary
        for (string& word : dictionary) {
            // Check if the word is a subsequence of s
           // and compare it with the current longest word based on length and lexicographical order
            if (isSubsequence(s, word) &&
                (longestWord.size() < word.size() ||
                (longestWord.size() == word.size() && word < longestWord))) {</pre>
                longestWord = word; // Update the longest word
        return longestWord; // Return the longest word
    // Helper function to check if string b is a subsequence of string a
    bool isSubsequence(string& a, string& b) {
        int aLength = a.size(), bLength = b.size(); // Length of the strings
        int i = 0, j = 0; // Pointers for each string
        while (i < aLength && j < bLength) {</pre>
            // If the characters match, increment j to check the next character of b
```

## // Always move the string pointer to the next character. stringPointer++;

```
// If no word from the dictionary can be formed by `s`, return an empty string.
      return '';
class Solution:
   def findLongestWord(self, s: str, dictionary: List[str]) -> str:
       # Helper function to check if b is a subsequence of a
       def is_subsequence(a, b):
            m, n = len(a), len(b)
            pos_a = pos_b = 0
           # Traverse both strings and check if b is subsequence of a
            while pos a < m and pos b < n:</pre>
               if a[pos a] == b[pos b]:
                    pos_b += 1 # Move pointer of string b if characters match
               pos_a += 1 # Always move pointer of string a
           # Check if reached the end of string b, meaning b is a subsequence of a
            return pos_b == n
       # Initialize the answer to an empty string
        longest_word = ''
       # Iterate over each word in the given dictionary
        for word in dictionary:
           # Check if the word is a subsequence of s
            # Update longest_word if word is longer or lexicographically smaller
            if is_subsequence(s, word) and (len(longest_word) < len(word) or</pre>
                                            (len(longest_word) == len(word) and longest_word > word)):
                longest_word = word
       # Return the longest word that is a subsequence of s and satisfies the condition
       return longest_word
Time and Space Complexity
  The given Python code defines a function findLongestWord that looks for the longest string in the dictionary that can be formed
  by deleting some of the characters of the string s. If there are more than one possible results, it returns the smallest in
  lexicographical order.
```

## Time Complexity: 0(n\*m + n\*log(n))Here n is the size of dictionary and m is the length of the string s. • The function check(a, b) has a time complexity of O(m), because in the worst case, it will check each character in string s against b.

**Time Complexity** 

• This check function is called for every word in the dictionary, resulting in O(n\*m).

- Additionally, sorting the dictionary in lexicographic order is required to ensure we get the smallest word when lengths are equal. Sorting a list of strings takes 0(n\*log(n)\*k), where k is the average length of strings; however, since we're not sorting the dictionary, we're not including this in our complexity analysis.
- **Space Complexity Space Complexity: 0(1)**
- No additional space is needed that grows with the input size. Only variables for iterating and comparison are used which occupy constant space. • Thus, the space complexity is constant since the only extra space used is for pointer variables i and j, and variable ans.