

[Articles](#) > [392. Is Subsequence](#) ▼[Previous \(/articles/repeated-substring-pattern/\)](#) Next [Next \(/articles/top-k-frequent-elements/\)](#)

## 392. Is Subsequence [\(/problems/is-subsequence/\)](/problems/is-subsequence/)

May 19, 2020 | 14.4K views

Average Rating: 4.74 (34 votes)

Given a string **s** and a string **t**, check if **s** is subsequence of **t**.

A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, "ace" is a subsequence of "abcde" while "aec" is not).

### Follow up:

If there are lots of incoming S, say S1, S2, ... , Sk where k >= 1B, and you want to check one by one to see if T has its subsequence. In this scenario, how would you change your code?

### Credits:

Special thanks to @pbrother (<https://leetcode.com/pbrother/>) for adding this problem and creating all test cases.

### Example 1:

**Input:** s = "abc", t = "ahbgdc"  
**Output:** true

### Example 2:

**Input:** s = "axc", t = "ahbgdc"  
**Output:** false

### Constraints:

- 0 <= s.length <= 100
- 0 <= t.length <= 10<sup>4</sup>

- Both strings consists only of lowercase characters.

## Solution

### Overview

First of all, one might be deceived by the **Easy** tag of the problem. The solution might be simple (đồng ý nếu xét về số dòng code), yet the problem itself is much more intriguing, especially when one is asked to prove the correctness of the solution, not to mention that we have an interesting and legitimate follow-up question.

Also, one might be puzzled with the hints from the problem description, which says *ÍMỞ ĐẦU*, *Ý NGHĨA CỦA VIỆC ĐẶT TÊN* and *ÍMỞ ĐẦU*. There is no doubt that each of them does characterize some trait of the solutions, although the order of these keywords might be misleading. Arguably, the keyword **Greedy** is more important than the other two.

One will see in the following sections, how each of the above **techniques** plays out in the solutions. We will also cover the follow-up question in one of the solutions.

### Approach 1: Divide and Conquer with Greedy

#### Intuition

The problem concerns the string matching issues, for which often one can apply a technique called **divide and conquer**.

The general idea of the divide and conquer technique is to reduce the problem down into subproblems with smaller scales (ví dụ như vậy) until the problem becomes small enough to tackle with. We then use the results of subproblems to construct the solution for the original problem.

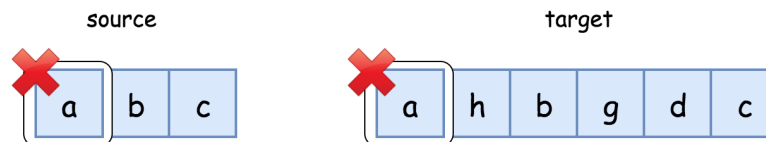
For more details, one can refer to the chapter of divide and conquer (<https://leetcode.com/explore/learn/card/recursion-ii/470/divide-and-conquer/>) in our Explore card.

Here we show how to break down our problem step by step. Given two strings `source` and `target`, we are asked to determine if the `source` string is a subsequence of the `target` string, `isSubsequence(source, target)`

Let us start from the first characters of each string,  $source[0]$ ,  $target[0]$ . By comparing them, there could be two cases as follows:

Case 1): they are identical,  $source[0] = target[0]$

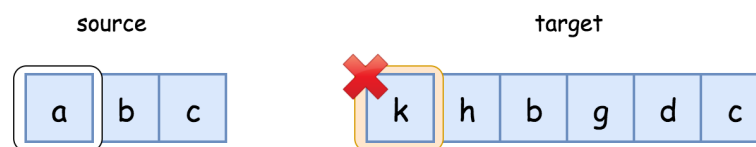
In this case, the best strategy would be to **cross out** the first characters in both strings, and then continue with the matching job.



By the above action, we reduce the input into a smaller scale. It boils down to determine if the rest source string ( $source[1:]$ ) is a subsequence of the rest of target string ( $target[1:]$ ), which we could summarize in the following recursive formula:

$$isSubsequence(source, target) = isSubsequence(source[1:], target[1:])$$

Case 2): they are not identical,  $source[0] \neq target[0]$



In this case, the only thing we can do is to **skip** (cross out) the first character in the target string, and keep on searching in the target string in the hope that we would find a letter that could match the first character in the source string.

Now, it boils down to determine if the source string could be a subsequence for the rest of the target string, which we summarize as follows:

$$isSubsequence(source, target) = isSubsequence(source, target[1:])$$

Let us combine the above two cases as follows, which consists of our baby steps of  $isSubsequence(source, target)$  by looking at the first characters of each string.

$$isSubsequence(source, target) = \begin{cases} isSubsequence(source[1:], target[1:]) \\ isSubsequence(source, target[1:]) \end{cases}$$

It should be intuitive to implement a recursive solution with the above formulas.

To make the recursion complete, we should also define the **base cases** properly. In this problem, we have two particular base cases:

- The **source** becomes empty, if we found matches for all the letters in the source string. Hence, the source string is a subsequence of the target string.
- The **target** becomes empty, if we exhaust the target string, yet there are still some letters left unmatched in the source string. Hence, the source string is not a subsequence of the target string.

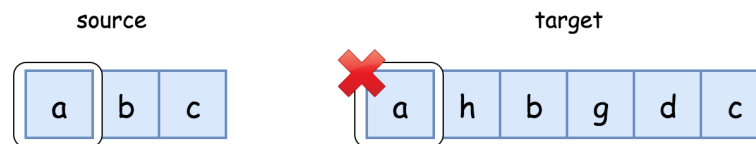
### Greedy

Before jumping into the implementation, we would like to discuss an important strategy that we adopted here, other than the **Key-Value** or **Map** technique.

That is right. It is the **Greedy** strategy, which we did not mention in the intuition section.

As one might recall, when the first characters of the source and target strings match, we mentioned that the **best strategy** is to **use** the matched characters from both strings and then continue with the matching.

The other possible action could be that we **dismiss** this match and continue the search in the target string.



By adopting the best strategy, we were **greedily** crossing out the matched character from the source string, rather than deferring the match.

One might question if the result is correct with the **brute force** strategy, since it does seem that we were missing out some other alternatives.

To prove the **correctness** of greedy algorithms, often we apply the contradiction technique, **by** deriving a contradicted fact while assuming the alternative argument is correct.

It could be tedious to give a rigid mathematical proof on why the greedy algorithm is correct here. Here we would like to present simply two arguments without detailed proofs:

- If the source is **not** a subsequence of the target string, in no case will our greedy algorithm return a positive result.

- If the source is indeed a subsequence of the target string (there could exist multiple solutions), then our greedy algorithm will return a positive result as well. For an obvious reason, our greedy algorithm does not exhaust all possible matches. However, one match suffices.


### Algorithm

Following the recursive formulas that we presented before, it should be intuitive to implement a solution with recursion.

As a reminder, here is the formulas:

$$\text{isSubsequence}(\text{source}, \text{target}) = \begin{cases} \text{isSubsequence}(\text{source}[1:], \text{target}[1:]) \\ \text{isSubsequence}(\text{source}, \text{target}[1:]) \end{cases}$$

Java
Python3

 Copy

```

1 class Solution {
2     String source, target;
3     Integer leftBound, rightBound;
4
5     private boolean rec_isSubsequence(int leftIndex, int rightIndex) {
6         // base cases
7         if (leftIndex == leftBound)
8             return true;
9         if (rightIndex == rightBound)
10            return false;
11
12        // consume both strings or just the target string
13        if (source.charAt(leftIndex) == target.charAt(rightIndex))
14            ++leftIndex;
15        ++rightIndex;
16
17        return rec_isSubsequence(leftIndex, rightIndex);
18    }
19
20    public boolean isSubsequence(String s, String t) {
21        this.source = s;
22        this.target = t;
23        this.leftBound = s.length();
24        this.rightBound = t.length();
25
26        return rec_isSubsequence(0, 0);
27    }

```

Note, the above implementations happen to comply with a particular form of recursion, called **tail recursion**, which is a runtime optimization technique that is supported in some programming languages such as C and C++.

For more details, one can refer to the article of tail recursion

(<https://leetcode.com/explore/learn/card/recursion-i/256/complexity-analysis/2374/>) in our Explore card.

### Complexity Analysis

Let  $|S|$  be the length of the source string, and  $|T|$  be the length of the target string.

- Time Complexity:  $\mathcal{O}(|T|)$ .

- At each invocation of the `rec_isSubsequence()` function, we would consume one character from the target string and optionally one character from the source string.
  - The recursion ends when either of the strings becomes empty. In the worst case, we would have to scan the entire target string. As a result, the overall time complexity of the algorithm is  $\mathcal{O}(|T|)$ .
  - Note, even when the source string is longer than the target string, the recursion would end anyway when we exhaust the target string. Hence, the number of recursions is not bounded by the length of the source string.
- Space Complexity:  $\mathcal{O}(|T|)$ 
    - The recursion incurs some additional memory consumption in the function call stack. As we discussed previously, in the worst case, the recursion would happen  $|T|$  times. Therefore, the overall space complexity is  $\mathcal{O}(|T|)$ .
    - With the optimization of tail recursion, this extra space overhead could be exempted, due to the fact that the call stack is reused for all consecutive recursions. However, Python and Java do not support tail recursion. Hence, this overhead cannot be waived.

## Approach 2: Two-Pointers

### Intuition

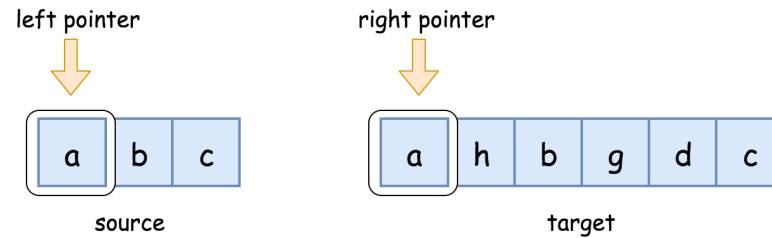
Following the same intuition above, we could further optimize the space complexity of the previous solutions, by replacing the recursion with the iteration.

More specifically, we iterate through the source and target strings, respectively with a **pointer**.

Each pointer marks a position that we progress on the matching of the characters.

### Algorithm

We designate two pointers for iteration, with the `left` pointer referring to the source string and the `right` pointer to the target string.



We move the pointers accordingly on the following two cases:

- If `source[left] == target[right]` : we found a match. Hence, we move both pointers one step forward.
- If `source[left] != target[right]` : no match is found. We then move only the right pointer on the target string.

The iteration would terminate, when either of the pointers exceeds its boundary.

At the end of the iteration, the result **solely** depends on the fact that whether we have consumed all the characters in the source string. If so, we have found a suitable match for each character in the source string. Therefore, the source string is a subsequence of the target string.

Java

Python3

```

1 class Solution {
2
3     public boolean isSubsequence(String s, String t) {
4         Integer leftBound = s.length(), rightBound = t.length();
5         Integer pLeft = 0, pRight = 0;
6
7         while (pLeft < leftBound && pRight < rightBound) {
8             // move both pointers or just the right pointer
9             if (s.charAt(pLeft) == t.charAt(pRight)) {
10                 pLeft += 1;
11             }
12             pRight += 1;
13         }
14         return pLeft == leftBound;
15     }
16 }
```

Copy

### Complexity Analysis

Let  $|S|$  be the length of the source string, and  $|T|$  be the length of the target string.

- Time Complexity:  $\mathcal{O}(|T|)$ 
  - The analysis is the same as the previous approach.
- Space Complexity:  $\mathcal{O}(1)$ 
  - We replace the recursion with iteration. In the iteration, a constant memory is consumed regardless of the input.

### Approach 3: Greedy Match with Character Indices Hashmap

#### Intuition

With the above two approaches under the belt, let us now look at the follow-up question raised in the problem description, which we cite as follows:

If there are lots of incoming  $S$ , say  $S_1, S_2, \dots$ , and you want to check one by one to see if  $T$  has its subsequence. In this scenario, how would you change your code?

In the above scenario, we would expect several incoming source strings, but a constant target string. We are asked to match each of the source strings against the target string.

If we apply our previous algorithms, for each match, the overall time complexity would be  $\mathcal{O}(|T|)$ .

In other words, regardless of the source strings, in the worst case, we have to scan the target string repeatedly, even though the target string remains the same.

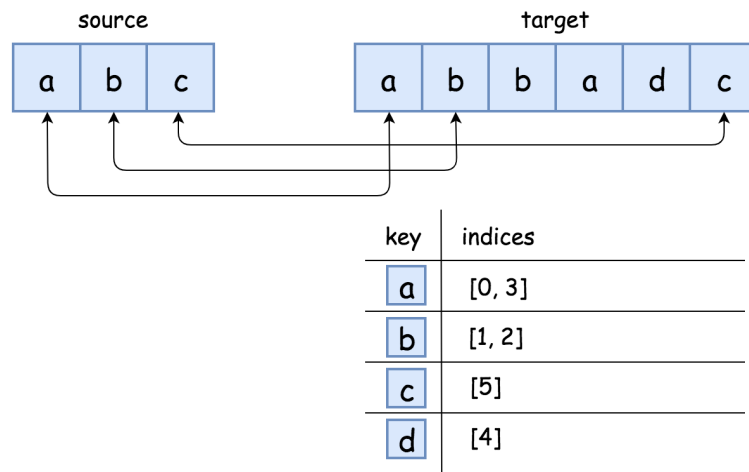
Now with the **Algorithm** identified, we could ask ourselves if we could do something about it.

The reason why we scan the target string is to **look for** the next character that matches a given character in the source string. In essence, this is a **lookup** operation in the array data structure.

To speed up the lookup operation, the data structure of **hashmap** could come in handy, since it has a  $\mathcal{O}(1)$  time complexity for its lookup operation.

Indeed, we could build a hashmap out of the target string, with each unique character as key and the indices of its appearance as value.





Moreover, we should pre-compute this hashmap once and then reuse it for all the following matches.

With this hashmap, rather than scanning through the entire target string, we could instantly retrieve all the relevant positions in the target string to look at, given a character from the source string.

### Algorithm

Essentially, the algorithm with hashmap remains rather similar with our previous approaches, as we still need to iterate through the source string to find the matches, and more importantly, we still do the match in the *book* manner.

- First, we build a hashmap out of the target string. Each key is a unique character in the target string, like `a`. Its corresponding value would be a list of indices where the character appears in the target string, like `[0, 3]`.
- We then iterate through the source string.
- This time, rather than keeping two pointers, we need only one pointer on the `target` string. The pointer marks our progress on the target string.
- As we've seen from all the previous approaches, the pointer on the target string should move **monotonically**, or in no case, we would move the pointer to an earlier position.
- We use the pointer to check if an index is suitable or not. For instance, for the character `a` whose corresponding indices are `[0, 3]`, we need to pick an index out of all the appearances as a match. Suppose at certain moment, the pointer is located at the index `1`. Then, the suitable *book* match would be the index of `3`, which is the first index that is larger than the current position of the target pointer.

Java

Python3

Copy

```

1 class Solution {
2
3     public boolean isSubsequence(String s, String t) {
4
5         // precomputation, build the hashmap out of the target string
6         HashMap<Character, List<Integer>> letterIndicesTable = new HashMap<>();
7         for (int i = 0; i < t.length(); ++i) {
8             if (letterIndicesTable.containsKey(t.charAt(i)))
9                 letterIndicesTable.get(t.charAt(i)).add(i);
10            else {
11                ArrayList<Integer> indices = new ArrayList<Integer>();
12                indices.add(i);
13                letterIndicesTable.put(t.charAt(i), indices);
14            }
15        }
16
17        Integer currMatchIndex = -1;
18        for (char letter : s.toCharArray()) {
19            if (!letterIndicesTable.containsKey(letter))
20                return false; // no match, early exist
21
22            boolean isMatched = false;
23            // greedy match with linear search
24            for (Integer matchIndex : letterIndicesTable.get(letter)) {
25                if (currMatchIndex < matchIndex) {
26                    currMatchIndex = matchIndex;
27                    isMatched = true;

```

### Optimization

As one might notice, we added a last touch to the above algorithm to make it faster.

Given a list of indices for a matched character, in order to find the suitable index, we could simply do the **linear search** as we did in the above Java implementation.

Since the list of indices is ordered, due to the process of construction, we could also apply the **binary search** on the list to locate the desired index faster. As a comparison, we implemented this in the Python implementation.

Now hopefully, no one is puzzled with the keyword of **linear search** from the hints of the problem.

### Complexity Analysis

Let  $|T|$  be the length of the target string, and  $|S|$  be the length of the source string.

- Time Complexity:  $\mathcal{O}(|T| + |S| \cdot \log |T|)$ .
  - First of all, we build a hashmap out of the target string, which would take  $\mathcal{O}(|T|)$  time complexity. But if we redesign the API to better fit the scenario of the follow-up question, we should put the construction of the hashmap in the constructor of the class, which should be done only once. The cost of this construction would be amortized by the following calls of string matches.

- As the second part of the algorithm, we scan through the source string, and lookup the corresponding indices in the hashmap. The lookup operation in hashmap is constant. However, to find the suitable index would take either  $\mathcal{O}(|T|)$  with the linear search or  $\mathcal{O}(\log |T|)$  with the binary search. To summarize, this part would be bounded by  $\mathcal{O}(|S| \cdot \log |T|)$ .
- As one can see, the second part heavily depends on the distribution of the characters in the target string. If the characters are distributed evenly, the entries in the hashmap would have a shorter list of indices, which in return would shorten the search time. But in general, one could consider the approach with hashmap should be faster than the two-pointers approach, although their time complexities say otherwise.
- Space Complexity:  $\mathcal{O}(|T|)$ 
  - We built a hashmap that consists of the indices for each character in the target string. Hence, the size of values (indices) in hashmap would be  $|T|$ . In the worst case, we might have as many keys as the values, as each character corresponds to a unique index. In total, the space complexity of the hashmap would be  $\mathcal{O}(|T|)$ .

## Approach 4: Dynamic Programming

### Intuition

Based on the description of the problem, it reminds us of a well-known problem called Edit Distance (<https://leetcode.com/problems/edit-distance/>) on LeetCode, which is also known as Levenshtein distance ([https://en.wikipedia.org/wiki/Levenshtein\\_distance](https://en.wikipedia.org/wiki/Levenshtein_distance)).

The problem of [Is Subsequence](#) is to find the minimal number of edit operations to convert one string to another string. The permitted operations are [Insert](#), [Delete](#) and [Replace](#).

Intuitively, one can consider our subsequence problem as a simplified [Is Subsequence](#) with only the [Delete](#) operation.

The classic solution to solve the edit distance problem is to apply the **dynamic programming** technique. Similarly, we should be able to solve our problem here with dynamic programming as well.

### Algorithm

With dynamic programming, essentially we build a matrix ( `dp[row][col]` ) where each element in the matrix represents the maximal number of deletions that we can have between a prefix of source string and a prefix of the target string, namely `source[0:col]` and `target[0:row]`.

e	0	1	1	2	2	3
c	0	1	1	2	2	2
a	0	1	1	1	1	1
#	0	0	0	0	0	0
#	a	b	c	d	e	

In the above graph, we show an example of the dp matrix. For instance, for the element of  $dp[2][3]$  as we highlighted, it indicates that the maximal number of deletions (2) matches that we can have between the source prefix 'ac' and the target prefix 'abc' is 2.

Suppose that we can obtain this dp matrix, the problem becomes simple.

Once we have the dp matrix, it boils down to see if we can have as many deletions as the number of characters in the source string, or if we could consume the entire source string with the matches from the target string.

In other words, it suffices to examine the last row of the dp matrix ( $dp[\text{len}(\text{source})]$ ), to see if there is any number that is equal to the length of the source string.

In the following animation, we show how to calculate each element in the dp matrix.

e	0	1				
c	0	1	1			
a	0	1	1			
#	0	0	0	0	0	0
#	a	b	c	d	e	

There is no match -->  
 $dp[\text{row}][\text{col}] = \max(dp[\text{row} - 1][\text{col}], dp[\text{row}][\text{col} - 1])$



7 / 18

Since the problem of **Key-Value** is a hard one and so with its solution, we invite you to check out our article of edit distance (<https://leetcode.com/articles/edit-distance/>), which explains the solution in details.

Here we present a modified version of the edit distance solution, based on the above idea.

Java

Python3

Copy

```

4
5      Integer sourceLen = s.length(), targetLen = t.length();
6      // the source string is empty
7      if (sourceLen == 0)
8          return true;
9
10     int[][] dp = new int[sourceLen + 1][targetLen + 1];
11     // DP calculation, we fill the matrix column by column, bottom up
12     for (int col = 1; col <= targetLen; ++col) {
13         for (int row = 1; row <= sourceLen; ++row) {
14             if (s.charAt(row - 1) == t.charAt(col - 1))
15                 // find another match
16                 dp[row][col] = dp[row - 1][col - 1] + 1;
17             else
18                 // retrieve the maximal result from previous prefixes
19                 dp[row][col] = Math.max(dp[row][col - 1], dp[row - 1][col]);
20         }
21         // check if we can consume the entire source string,
22         // with the current prefix of the target string.
23         if (dp[sourceLen][col] == sourceLen)
24             return true;
25     }
26
27     // matching failure
28     return false;
29 }
30 }

```

### Complexity Analysis

Let  $|T|$  be the length of the target string, and  $|S|$  be the length of the source string.

- Time Complexity:  $\mathcal{O}(|S| \cdot |T|)$ 
  - We build a matrix of size  $|S| \cdot |T|$ . In the worst case, we would need to iterate through each element in the matrix. Therefore, the overall time complexity of the algorithm is  $\mathcal{O}(|S| \cdot |T|)$ .
  - It is not necessarily the case that we have to iterate through the entire matrix. As one notices, we do have an early exit condition while we fill the values in the matrix.
  - Generally speaking, the dynamic programming algorithms tend to be faster than other solutions, since it reuses the intermediate solutions rather than recalculating them. However, it is not the case here.
  - The DP solution here tries to calculate the solutions for all combinations of prefixes between the source and target strings. As a result, it is less efficient than the `isSubsequence` approach which has the time complexity of  $\mathcal{O}(|T|)$ .
- Space Complexity:  $\mathcal{O}(|S| \cdot |T|)$ , since we build a matrix of size  $|S| \cdot |T|$ .

Rate this article:

[Previous \(/articles/repeated-substring-pattern/\)](/articles/repeated-substring-pattern/)[Next \(/articles/top-k-frequent-elements/\)](/articles/top-k-frequent-elements/)Comments: **16**

Sort By ▼

Type comment here... (Markdown is supported)

Preview

Post

shuangpan (/shuangpan) ★ 15 ⌚ May 27, 2020 8:19 PM ⋮

Java two pointers 1ms. I don't know why others solutions are super complex, it is super easy to use two pointers.

[Read More](#)

8 ▲ ▼ Share Reply

SHOW 6 REPLIES

yoursungjin (/yoursungjin) ★ 29 ⌚ June 2, 2020 2:38 AM ⋮

there's another dp solution. you can get the length of the longest common sequence and compare it to the source length.

3 ▲ ▼ Share Reply

SHOW 3 REPLIES

WinningCombo (/winningcombo) ★ 2 ⌚ June 9, 2020 3:43 AM ⋮

For anyone trying to figure out how the `bisect.bisect_right` is working:

`bisect.bisect_right` gives the index to which an element is to be inserted so as to maintain the sorted order of the array. For example, `a = [1,3,5,7,9]` then `bisect.bisect_right(a, 6)` gives 3 as the output. However, it becomes a little tricky when the element that is being inserted is already present in the array. In such a case the element is inserted in the right most index possible. For instance `a = [1 3 5 7 9]` then `bisect.bisect_right(a, 7)` gives 4 as the

[Read More](#)

2 ▲ ▼ Share Reply

kakrafoon (/kakrafoon) ★ 7 ⌚ June 16, 2020 11:41 AM ⋮

Very instructive notes. The two pointer solution is the 'easy' part. Great job on bringing the connection with greedy and especially, dynamic programming.

1 ▲ ▼ Share Reply

victorcui96 (/victorcui96) ★ 15 ⌚ June 9, 2020 1:31 PM ⋮

For the DP approach, can someone explain the intuition why we use the `max()` function

```
dp[row][col] = max(dp[row][col - 1], dp[row - 1][col])
```