

Edit Distance

We'll cover the following ^

- Problem Statement
- Basic Solution
 - Code
- Top-down Dynamic Programming with Memoization
 - Code
- Bottom-up Dynamic Programming
 - Code

Problem Statement

Given strings s_1 and s_2 , we need to transform s_1 into s_2 by deleting, inserting, or replacing characters. Write a function to calculate the count of the minimum number of edit operations.

Example 1:

```
Input: s1 = "bat"
       s2 = "but"
Output: 1
Explanation: We just need to replace 'a' with 'u' to transform s1 to s2.
```

Example 2:

```
Input: s1 = "abdca"
       s2 = "cbda"
Output: 2
Explanation: We can replace first 'a' with 'c' and delete second 'c'.
```

Example 3:

```
Input: s1 = "passpot"
       s2 = "ppsspqrt"
Output: 3
Explanation: Replace 'a' with 'p', 'o' with 'q', and insert 'r'.
```

Basic Solution

A basic brute-force solution could be to try all operations (one by one) on each character of s_1 . We can iterate through s_1 and s_2 together. Let's assume $index_1$ and $index_2$ point to the current indexes of s_1 and s_2 respectively, so we have two options at every step:

1. If the strings have a matching character, we can recursively match for the remaining lengths.



2. If the strings don't match, we start three new recursive calls representing the three edit operations. Whichever recursive call returns the minimum count of operations will be our answer.

Code #

Here is recursive implementation:

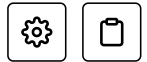
| | | | |
|------|----|---------|-----|
| Java | JS | Python3 | C++ |
|------|----|---------|-----|

```
1 def find_min_operations(s1, s2):
2     return find_min_operations_recursive(s1, s2, 0, 0)
3
4
5 def find_min_operations_recursive(s1, s2, i1, i2):
6
7     n1, n2 = len(s1), len(s2)
8     # if we have reached the end of s1, then we have to insert all the remaining characters
9     if i1 == n1:
10         return n2 - i2
11
12     # if we have reached the end of s2, then we have to delete all the remaining characters
13     if i2 == n2:
14         return n1 - i1
15
16     # If the strings have a matching character, we can recursively match for the remaining
17     if s1[i1] == s2[i2]:
18         return find_min_operations_recursive(s1, s2, i1 + 1, i2 + 1)
19
20     # perform deletion
21     c1 = 1 + find_min_operations_recursive(s1, s2, i1 + 1, i2)
22     # perform insertion
23     c2 = 1 + find_min_operations_recursive(s1, s2, i1, i2 + 1)
24     # perform replacement
25     c3 = 1 + find_min_operations_recursive(s1, s2, i1 + 1, i2 + 1)
26
27     return min(c1, min(c2, c3))
28
29
30 def main():
31     print(find_min_operations("bat", "but"))
32     print(find_min_operations("abdca", "cbda"))
33     print(find_min_operations("passpot", "ppsspqrt"))
34
35
36 main()
```

Because of the three recursive calls, the time complexity of the above algorithm is exponential $O(3^{m+n})$, where 'm' and 'n' are the lengths of the two input strings. The space complexity is $O(n + m)$ which is used to store the recursion stack.

Top-down Dynamic Programming with Memoization #

We can use an array to store the already solved subproblems.



The two changing values in our recursive function are the two indexes, $i1$ and $i2$. Therefore, we can store the results of all the subproblems in a two-dimensional array. (Another alternative could be to use a hash-table whose key would be a string ($i1 + "|" + i2$)).

Code #

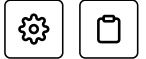
Here is the code for Top-down DP approach:

| Java | JS | Python3 | C++ |
|------|----|---------|-----|
|------|----|---------|-----|

```
1 def find_min_operations(s1, s2):
2     dp = [[-1 for _ in range(len(s2)+1)] for _ in range(len(s1)+1)]
3     return find_min_operations_recursive(dp, s1, s2, 0, 0)
4
5
6 def find_min_operations_recursive(dp, s1, s2, i1, i2):
7     n1, n2 = len(s1), len(s2)
8     if dp[i1][i2] == -1:
9         # if we have reached the end of s1, then we have to insert all the remaining character
10        if i1 == n1:
11            dp[i1][i2] = n2 - i2
12
13        # if we have reached the end of s2, then we have to delete all the remaining character
14        elif i2 == n2:
15            dp[i1][i2] = n1 - i1
16
17        # If the strings have a matching character, we can recursively match for the remaining
18        elif s1[i1] == s2[i2]:
19            dp[i1][i2] = find_min_operations_recursive(
20                dp, s1, s2, i1 + 1, i2 + 1)
21        else:
22            c1 = find_min_operations_recursive(
23                dp, s1, s2, i1 + 1, i2) # delete
24            c2 = find_min_operations_recursive(
25                dp, s1, s2, i1, i2 + 1) # insert
26            c3 = find_min_operations_recursive(
27                dp, s1, s2, i1 + 1, i2 + 1) # replace
28            dp[i1][i2] = 1 + min(c1, min(c2, c3))
29
30    return dp[i1][i2]
31
32
33 def main():
34     print(find_min_operations("bat", "but"))
35     print(find_min_operations("abdca", "cbda"))
36     print(find_min_operations("passpot", "ppsspqrt"))
37
38
39 main()
```

What is the time and space complexity of the above solution? Since our memoization array `dp[s1.length()][s2.length()]` stores the results for all the subproblems, we can conclude that we will not have more than $m * n$ subproblems (where 'm' and 'n' are the lengths of the

two input strings.). This means that our time complexity will be $O(m * n)$.



The above algorithm will be using $O(m * n)$ space for the memoization array. Other than that we will use $O(m + n)$ space for the recursion call-stack. So the total space complexity will be $O(m * n + (m + n))$, which is asymptotically equivalent to $O(m * n)$.

Bottom-up Dynamic Programming

Since we want to match all the characters of the given two strings, we can use a two-dimensional array to store our results. The lengths of the two strings will define the size of the two dimensions of the array. So for every index 'i1' in string 's1' and 'i2' in string 's2', we will choose one of the following options:

1. If the character `s1[i1]` matches `s2[i2]`, the count of the edit operations will be equal to the count of the edit operations for the remaining strings.
2. If the character `s1[i1]` does not match `s2[i2]`, we will take the minimum count from the remaining strings after performing any of the three edit operations.

So our recursive formula would be:

```
1 if s1[i1] == s2[i2]
2     dp[i1][i2] = dp[i1-1][i2-1]
3 else
4     dp[i1][i2] = 1 + min(dp[i1-1][i2], // delete
5                           dp[i1][i2-1], // insert
6                           dp[i1-1][i2-1]) // replace
```

Code

Here is the code for our bottom-up dynamic programming approach:

| Java | JS | Python3 | C++ |
|---|----|---------|-----|
| <pre>1 def find_min_operations(s1, s2): 2 n1, n2 = len(s1), len(s2) 3 dp = [[-1 for _ in range(n2+1)] for _ in range(n1+1)] 4 5 # if s2 is empty, we can remove all the characters of s1 to make it empty too 6 for i1 in range(n1+1): 7 dp[i1][0] = i1 8 9 # if s1 is empty, we have to insert all the characters of s2 10 for i2 in range(n2+1): 11 dp[0][i2] = i2 12 13 for i1 in range(1, n1+1): 14 for i2 in range(1, n2+1): 15 # If the strings have a matching character, we can recursively match for the remaini 16 if s1[i1 - 1] == s2[i2 - 1]: 17 dp[i1][i2] = dp[i1 - 1][i2 - 1] 18 else: 19 dp[i1][i2] = 1 + min(dp[i1 - 1][i2], # delete 20 min(dp[i1][i2 - 1], # insert 21 dp[i1 - 1][i2 - 1])) # replace 22 23 return dp[n1][n2] 24</pre> | | | |

```

25
26 def main():
27     print(find_min_operations("bat", "but"))
28     print(find_min_operations("abdca", "cbda"))
29     print(find_min_operations("passpot", "ppsspqrt"))
30
31
32     main()
33

```



The time and space complexity of the above algorithm is $O(n * m)$, where 'm' and 'n' are the lengths of the two input strings.

← Back

Next →

Longest Alternating Subsequence

Strings Interleaving

☒ Mark as Completed



Report an Issue



Ask a Question

(https://discuss.educative.io/tag/edit-distance__pattern-5-longest-common-substring__grokking-dynamic-programming-patterns-for-coding-interviews)

Could not connect to the reCAPTCHA service. Please check your internet connection and reload to get a reCAPTCHA challenge.