Edit Distance

We'll cover the following

Problem Statement

Given strings s1 and s2, we need to transform s1 into s2 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 s1. We can iterate through s1 and s2 together. Let's assume index1 and index2 point to the current indexes of s1 and s2 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++

```
def find_min_operations(s1, s2):
  return find_min_operations_recursive(s1, s2, 0, 0)


def find_min_operations_recursive(s1, s2, i1, i2):

  n1, n2 = len(s1), len(s2)
  # if we have reached the end of s1, then we have to insert all the remaining characters
  if i1 == n1:
    return n2 - i2

  # if we have reached the end of s2, then we have to delete all the remaining characters
  if i2 == n2:
    return n1 - i1

  # If the strings have a matching character, we can recursively match for the remaining l
  if s1[i1] == s2[i2]:
    return find_min_operations_recursive(s1, s2, i1 + 1, i2 + 1)

  # perform deletion
  c1 = 1 + find_min_operations_recursive(s1, s2, i1 + 1, i2)
  # perform insertion
  c2 = 1 + find_min_operations_recursive(s1, s2, i1, i2 + 1)
  # perform replacement
  c3 = 1 + find_min_operations_recursive(s1, s2, i1 + 1, i2 + 1)

  return min(c1, min(c2, c3))


def main():
  print(find_min_operations("bat", "but"))
  print(find_min_operations("abdca", "cbda"))
  print(find_min_operations("passpot", "ppsspqrt"))


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++

```python
def find_min_operations(s1, s2):
  dp = [[-1 for _ in range(len(s2)+1)] for _ in range(len(s1)+1)]
  return find_min_operations_recursive(dp, s1, s2, 0, 0)


def find_min_operations_recursive(dp, s1,  s2,  i1,  i2):
  n1, n2 = len(s1), len(s2)
  if dp[i1][i2] == -1:
    # if we have reached the end of s1, then we have to insert all the remaining  characte
    if i1 == n1:
      dp[i1][i2] = n2 - i2

    # if we have reached the end of s2, then we have to delete all the remaining character
    elif i2 == n2:
      dp[i1][i2] = n1 - i1

    # If the strings have a matching character, we can recursively match for the remaining
    elif s1[i1] == s2[i2]:
      dp[i1][i2] = find_min_operations_recursive(
        dp, s1, s2, i1 + 1, i2 + 1)
    else:
      c1 = find_min_operations_recursive(
        dp, s1, s2, i1 + 1, i2)  # delete
      c2 = find_min_operations_recursive(
        dp, s1, s2, i1, i2 + 1)  # insert
      c3 = find_min_operations_recursive(
        dp, s1, s2, i1 + 1, i2 + 1)  # replace
      dp[i1][i2] = 1 + min(c1, min(c2, c3))

  return dp[i1][i2]


def main():
  print(find_min_operations("bat", "but"))
  print(find_min_operations("abdca", "cbda"))
  print(find_min_operations("passpot", "ppsspqrt"))


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:

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

Code

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

Java | JS | Python3 | C++

```python
def find_min_operations(s1, s2):
  n1, n2 = len(s1), len(s2)
  dp = [[-1 for _ in range(n2+1)] for _ in range(n1+1)]

  # if s2 is empty, we can remove all the characters of s1 to make it empty too
  for i1 in range(n1+1):
    dp[i1][0] = i1

  # if s1 is empty, we have to insert all the characters of s2
  for i2 in range(n2+1):
    dp[0][i2] = i2

  for i1 in range(1, n1+1):
    for i2 in range(1, n2+1):
      # If the strings have a matching character, we can recursively match for the remaini
      if s1[i1 - 1] == s2[i2 - 1]:
        dp[i1][i2] = dp[i1 - 1][i2 - 1]
      else:
        dp[i1][i2] = 1 + min(dp[i1 - 1][i2],  # delete
                             min(dp[i1][i2 - 1],  # insert
                                 dp[i1 - 1][i2 - 1]))  # replace

  return dp[n1][n2]

```

```

def main():
  print(find_min_operations("bat", "but"))
  print(find_min_operations("abdca", "cbda"))
  print(find_min_operations("passpot", "ppsspqrt"))


main()

```

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

Longest Alternating Subsequence

Next →

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)