Project Report

Sandro’s Biography

Thitare Nimanong (6210015)

Ornwara Sangthongnirundorn (6215112)

CSX3009 Algorithm Design (1/2021)

Project title: Sandro’s Biography (Timus #1786, Difficulty: 101)

Table Of Contents

Sandro's Biography

Chapter 1: Problem

1.1 Problem Statement

The full details of the problem can be found at:

https://acm.timus.ru/problem.aspx?space=1&num=1786.

Scenario

"A magician has come across a set of ancient scrolls in the library. He wishes to modify the letters in the scrolls so that the name "Sandro" appears in them. To do so, he will hire a corrector. Two possible modifications can be made: changing a letter in the scroll for another of the same case (uppercase letter to uppercase letter or lowercase letter to lowercase letter) or changing the case of a letter (uppercase to lowercase or vice versa). Each change will cost 5 gold coins each. What is the minimum cost of a modification to have the name "Sandro" appear in the scrolls? [1]"

Input

A single line of string input, *A*, consisting of both uppercase and lowercase English letters where 6 <= len(A) <= 200.

Output

The minimum amount of gold coins must be paid to the collector to make the name "Sandro" appear in the manuscript.

Sample

Input = “MyNameisAlexander”

Output = 20

1.2 Breakdown

Given input A and string B, A[i] can be changed for another letter of the same case as B[i] or switch its case for the case of B[i].

For each letter in A, there are three possible cases with two conditions:

1. A[i] is the same letter as B[i] but of a different case, *e.g. "a" and "A"*.
2. A[i] is the same case as B[i] but a different letter, *e.g. "A" and "B"*.
3. A[i] is a different letter and a different case from B[i], *e.g. "a" and "B"*.

If A[i] and B[i] is the same letter but in a different case, only one modification is needed: changing the case (5 gold coins).

If A[i] and B[i] are different letters of the same case, only one modification is needed: changing the letter (5 gold coins).

However, if A[i] and B[i] are different letters of different cases, two modifications are needed: changing the letter and the case (5 gold coins * 2 = 10 gold coins).

Given the input "MyNameisAlexander", the minimum amount that must be paid is 20 gold coins. This means that the string needs a minimum of 4 modifications to have the name "Sandro" appear in the manuscript.

1.3 Different Approaches to The Problem

This problem is essentially an "Edit Distance" task. To solve this, we can use Levenshtein's algorithm. However, in this project, we used a brute force approach.

Our first implementation of the solution uses the brute-force approach.

Chapter 2: Implementation

Our initial approach was to use a brute-force solution: loop through all the characters of both strings and check the operations required to match the character, case, or both.

First, for each character in the input string (“MyNameisAlexander”), we compare the case and character to every character in “Sandro”. We initialize a variable “count” to 0 to keep track of the total amount of changes. Another global variable “mcount” holds the current minimum number of changes the string needs.

Characters with the same case only need to be exchanged for another character, and the same characters with different cases only need their cases to be switched. In either case, the “count” is increased by 1. If a character is different in both the letter and the case, then both must be changed, and the “count” variable is incremented by two.

After the nested for loop is completed, the variable “mcount” is reset to the lowest value between its current “mcount” value and the latest “count” value.

Finally, to return the total minimum cost in gold coins, we return the variable “mcount” multiplied by 5 because each change costs five gold coins each.

```python
import sys, time
sys.setrecursionlimit(1000000000)

n = input()
k = "Sandro"
remain = len(n) - len(k)
mcount = 999999999
start = time.time()
for i in range(len(n)):
    t = 0
    count = 0
    if i > remain:
        break
    else:
        for j in range(i, i + len(k)):
            if n[j] != k[t]:
                if n[j].isupper() == k[t].isupper() or n[j].islower() == k[t].islower(): #check case
                    count += 1
                elif n[j].upper() == k[t].upper() or n[j].lower() == k[t].lower(): #check same character
                    count += 1
                else:
                    count += 2
            t += 1
        mcount = min(mcount, count)
end = time.time()
print(mcount * 5)
elapsed_time = "{:.20f}".format(end-start)
print(f"elapsed: {elapsed_time}")
```

Figure 0.1 Initial approach: Brute-force

While this may not be the optimal solution, the brute-force approach was fast enough to pass the Timus Online Judge. The time complexity of this solution is O(kn), where “k” is the length of the desired string (“Sandro”) and “n” is the length of the input string (“MyNameisAlexander”). Because we do not use memoization, the space complexity of the algorithm itself is lower than the dynamic programming technique which has a space complexity of O(kn). Instead, we store only the two strings of lengths k and n, and four int variables (“remain”, “mcount”, “count”, and “t”).

Another implementation is to use recursion rather than multiple for-loops. In this implementation, we created another function to hold the repetitive steps in the nested for loop.

The function levenD(i) takes a single parameter, i, to be used as the index in place of the “for i in range(len(n))” statement. After each call to checkCase(i,t,i+len(k)), levenD(i) recursively calls itself with an incremented i value (levenD(i+1)).

```python
import sys, time
sys.setrecursionlimit(1000000000)

n = input()
k = "Sandro"
remain = len(n) - len(k)
mcount = 999999999
count = 0

def levenD(i):
    global n, k, remain, mcount, count
    if i < len(n):
        t = 0
        count = 0
        if i > remain:
            return -1
        else:
            checkCase(i,t, i+len(k))
            mcount = min(mcount, count)
    levenD(i+1)

def checkCase(j,t,y):
    global count
    if j >= y:
        return -1
    else:
        if k[t] != n[j]:
            if n[j].isupper() == k[t].isupper() or n[j].islower() == k[t].islower():
                count += 1

            elif n[j].upper() == k[t].upper():
                count += 1
            else:
                count += 2
    checkCase(j+1,t+1,y)
start = time.time()
levenD(0)
end = time.time()
print(mcount*5)

elapsed_time = "{:.20f}".format(end-start)
print(f"elapsed: {elapsed_time}")
```

Figure 0.2 Using recursion to optimize the initial solution

The function checkCase(i, t, i + len(k)) checks all the conditions (letter and case) of each letter in the input and Sandro. Depending on the condition, count is incremented by either 1 (only one change needed to either the letter or the case), or 2 to signify a change in both the letter and the case.

Comparison of the initial approach and the use of recursion

After comparing the runtimes of both approaches, the initial approach using for-loops is faster than the recursive version.

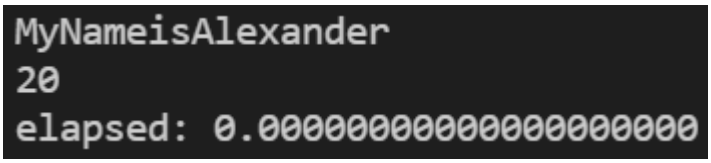

Figure 0.3 Runtime of Initial Approach

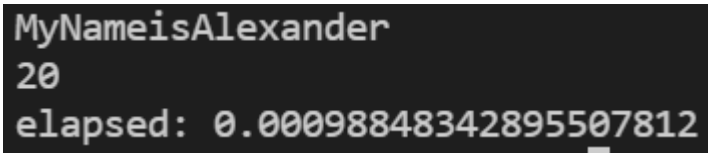

Figure 0.4 Runtime of Recursive version

Solutions judgement results

ID	Date	Author	Problem	Language	Judgement result	Test #	Execution time	Memory used
9456707	14:56:42 10 Sep 2021	Ornwara	1786. Sandro's Biography	Python 3.8 x64	Accepted		0.078	400 KB
9456706	14:56:12 10 Sep 2021	Ornwara	1786. Sandro's Biography	Python 3.8 x64	Accepted		0.078	472 KB

Figure 0.5 Submitting both versions on Timus Online Judge

As seen in Figure 0.5, we submitted both solutions on Timus Online Judge. While both implementations passed, the first implementation using for-loops (ID = 9456707) uses less memory than the second implementation using recursive calls (ID = 9456706).

Other Solutions

As mentioned earlier, the most popular solution to this problem uses Levenshtein's Distance algorithm.

```
# A Naive recursive Python program to find minimum number
# operations to convert str1 to str2
def editDistance(str1, str2, m, n):

    # If first string is empty, the only option is to
    # insert all characters of second string into first
    if m == 0:
         return n

    # If second string is empty, the only option is to
    # remove all characters of first string
    if n == 0:
        return m

    # If last characters of two strings are same, nothing
    # much to do. Ignore last characters and get count for
    # remaining strings.
    if str1[m-1]== str2[n-1]:
        return editDistance(str1, str2, m-1, n-1)

    # If last characters are not same, consider all three
    # operations on last character of first string, recursively
    # compute minimum cost for all three operations and take
    # minimum of three values.
    return 1 + min(editDistance(str1, str2, m, n-1),    # Insert
                   editDistance(str1, str2, m-1, n),    # Remove
                   editDistance(str1, str2, m-1, n-1)    # Replace
                   )

# Driver program to test the above function
str1 = "sunday"
str2 = "saturday"
print editDistance(str1, str2, len(str1), len(str2))
```

Figure 0.6 Levenshtein's Distance Algorithm [2]

The code shown in Figure 0.6 is GeekForGeek's implementation of Levenshtein's Distance algorithm. The editDistance function has two base cases; if either str1 or str2 is empty, the function returns the length of the other string because it will need to insert every letter into the other string.

If the last characters of both strings are the same, the function recursively calls itself, leaving out the last character of each string. It does not need to do any further functions on them because they are the same. Otherwise, it recursively calls itself again three times, once for each operation, and adds 1 and the minimum value of all three operations to get the lowest count possible.

This implementation uses dynamic programming, finding the best solution for each overlapping subproblem to build up to the optimal solution [3].

To better understand the mechanisms of this approach, we can use a table to map the outputs.

The first row and column are filled by numbers between 0 and n, where n is the length of the string. This is because the empty string must go through n insertions to be equal to the other string. These figures serve as the base case.

From there, for each new item, we evaluate the minimum value from the boxes immediately next to it. The highlighted cells are those immediately next to it. If the characters in the corresponding row and column are different, its cell is filled with the minimum value + 1 [4].

	""	x	a	n	d	e	r
""	0	1	2	3	4	5	6
S	1	1					
a	2						
n	3						
d	4						
r	5						
o	6						

Figure 0.7 Levenshtein's Distance Trace Table

Figure 0.7 shows the trace table when evaluating what value should fill the cell at “x” and “S”. Because the two letters are different, the value filled in the cell is the minimum value out of the three highlighted cells (1, 0, 1) add 1. The lowest value is 0, so the cell of “x” and “S” is filled by 1. If the characters are the same, the new value to be inserted is the same as the minimum value. This continues until all the cells are filled. The value in the last cell is the optimal solution.

	""	x	a	n	d	e	r
""	0	1	2	3	4	5	6
S	1	1	2	3	4	5	6
a	2	2	1	2	3	4	5
n	3	3	3	2	3	4	5
d	4	4	4	3	2	3	4
r	5	5	5	4	3	3	3
o	6	6	6	5	4	4	4

Figure 0.8 Complete trace table

The time complexity of this implementation is $O(3^n)$ because each subproblem is called three times. This can be improved by using memoization techniques as shown in Figure 0.9. With memoization, both the time and space complexity are O(m*n).

```python
# A memoization program to find minimum number
# operations to convert str1 to str2
def editDistance(str1, str2, m, n, d = {}):

    key = m, n

    # If first string is empty, the only option
    # is to insert all characters of second
    # string into first
    if m == 0:
        return n

    # If second string is empty, the only
    # option is to remove all characters
    # of first string
    if n == 0:
        return m

    if key in d:
        return d[key]

    # If last characters of two strings are same,
    # nothing much to do. Ignore last characters
    # and get count for remaining strings.
    if str1[m - 1] == str2[n - 1]:
        return editDistance(str1, str2, m - 1, n - 1)

    # If last characters are not same, consider
    # all three operations on last character of
    # first string, recursively compute minimum
    # cost for all three operations and take
    # minimum of three values.

    # Store the returned value at dp[m-1][n-1]
    # considering 1-based indexing
    d[key] = 1 + min(editDistance(str1, str2, m, n - 1), # Insert
                     editDistance(str1, str2, m - 1, n), # Remove
                     editDistance(str1, str2, m - 1, n - 1)) # Replace
    return d[key]

# Driver code
str1 = "sunday"
str2 = "saturday"

print(editDistance(str1, str2, len(str1), len(str2)))

# This code is contributed by puranjanprithu
```

Figure 0.9 Memoized version of Levenshtein's Distance Algorithm [2]

References

[1] Olga Soboleva (Ural Regional School Programming Contest 2010) *1786. Sandro's Biography* https://acm.timus.ru/problem.aspx?space=1&num=1786

[2] Striver (28 May 2021) *Edit Distance | DP using Memoization* GeeksForGeeks. https://www.geeksforgeeks.org/edit-distance-dp-using-memoization/

[3] The EdPresso Team *The Levenshtein distance algorithm* https://www.educative.io/edpresso/the-levenshtein-distance-algorithm

[4] Back To Back SWE (11 January 2019) *Edit Distance Between 2 Strings - The Levenshtein Distance ("Edit Distance" on LeetCode).* https://www.youtube.com/watch?v=MiqoA-yF-0M