Universidad EAFIT

Departamento de Informática y Sistemas

Sentence Decomposition: Analyzing the problem

Data Structures and Algorithms II

Author:
Simón Marín Giraldo
May 2020

Contents

1 Introduction

1.1 Problem definition

We are given a String problem in which we are requested to minimize the cost of operations over sequences of characters.
We receive a list of valid words and a sentence that can contain the accepted words from the list. This sentence contains each word from the list but their characters are mixed. Anyway, the order in which the mixed words appear is the same as the order of the words in the dictionary.

For example:

We have the sentence: **“neotowheret”**
And we have the dictionary: **{“one”, “two”, “three”, “there”}**
Even if we have the sentence in disorder, the order of the dictionary is kept:

The underlined part in the sentence:

$$\textbf{“}\underline{\textbf{neo}}\textbf{towheret”}$$

Matches with the first word in the dictionary:

$$\textbf{\{“}\underline{\textbf{one}}\textbf{”, “two”, “three”, “there”\}}$$

Then we have the second visible valid word:

$$\textbf{“neo}\underline{\textbf{tow}}\textbf{heret”}$$

It matches with the second word in the dictionary:

$$\textbf{\{“one”, “}\underline{\textbf{two}}\textbf{”, “three”, “there”\}}$$

Let’s now take a look at the two last elements in our dictionary:

$$\textbf{\{“one”, “two”, “}\underline{\textbf{three}}\textbf{”, “}\underline{\textbf{there}}\textbf{”\}}$$

As we can notice, with the last five characters in our sentence, we can build any of the last two words in the dictionary.

$$\textbf{“neotow}\underline{\textbf{heret}}\textbf{”}$$

With the following definitions and the solution explanation, we will determine how to optimally minimize the cost of transformation and solve the problem.

1.1.1 Concepts

- ***Cost***: it is defined as the number of characters in which two given Strings differ.

For example:

Let's consider the following Strings:

"neo"
"one"

As explained in the previous definition, cost is the number of character positions in which two Strings differ. In this case, the first character of the first String is "n". Now, let's look at the first character in the second String. It is "o". As "n" $\neq$ "o", we proceed to add 1 to the cost.

Now, let's take a look at the second character in each String. For the first String's second character we have "e". Now, for the second String's second character, we have "n". As "e" $\neq$ "n", we add 1 to the cost. Now, the cost has a value of 2 because we have found two character positions in which both Strings differ.

Finally, let's see the third and last character in both Strings. For the first String's third character we have "o". Now, for the second's String third character, we have "e". As "o" $\neq$ "e", we add 1 to the cost. Now the total cost has a value of 3 because we found three character positions in which both Strings differ.

- ***Anagram***: Two words are anagrams when they both have the same length and the same quantity of each character.

  For example:

  Let's consider the following Strings:

  "three"
  "heret"

  As explained in the previous definition, anagrams are words that have both the same length and the same quantity of each character.
  First of all, we need to make sure that both words have the same length:

$$|“three''| = 5$$
$$|“heret''| = 5$$

  As we can see, both Strings have the same length. Now, we start counting the quantity of each character:

$$|“three''|_t = 1$$
$$|“heret''|_t = 1$$

As we can see, we have the same number of “t” characters in both Strings. We continue with all the other characters:

$$\begin{gathered}|\text{“}three''|_h = 1\\ |\text{“}heret''|_h = 1\\ |\text{“}three''|_r = 1\\ |\text{“}heret''|_r = 1\\ |\text{“}three''|_e = 2\\ |\text{“}heret''|_e = 2\end{gathered}$$

As we can see, every character in “three” is present in “heret” in the same quantity.

A more formal definition for anagrams would be:

Let Σ be an alphabet.
Let x, y be some words with Σ symbols.
Let $|x|_a$ be the total amount of a’s in x i.e: $|aab|_a = 2, |aab|_c = 0$.

Now, we can define
A: $word \times word \to \{true, false\}$ as a function such that:

$$A(x,y) = \begin{cases} true, & \text{if } |x| = |y| \text{ and } |x|_\delta = |y|_\delta, \forall_\delta \in \Sigma \\ false, & \text{otherwise} \end{cases}$$

1.2 Input

- ***Sentence: String***: This parameter is a String which is going to be analyzed and compared with words in a dictionary.
- ***Dictionary: String array***: This parameter is the dictionary which will be used to analyze the sentence.

1.3 Output

The program will return an integer that will represent the minimum total cost that will be needed to turn the given sequence into a sequence with the accepted words which are specified in the dictionary. If there is no such possible transformation that will allow us to convert the given sequence into a sequence made from the accepted words, the result to be returned will be -1.

2 Solution and complexity analysis

Now, we are going to describe the coding solution given to this particular problem. We must take into account that this solution must work for every case for it to be a valid solution accepted by the online judge.

2.1 Finding anagrams

We previously defined what was an anagram and a method for finding them. Since we are concerned about optimizing the execution times and algorithmic complexity, we must have an optimal method to determine if two words are anagrams. Instead of counting each type of character in a word, checking with the next ones and comparing each character length, which has a complexity of:

$$T(n)_1 = \underbrace{nk}_{x} + \underbrace{nk}_{y}$$

$$T(n)_1 = 2nk$$

Where n is the length of the word and k is the number of symbols in the alphabet Σ.

The final Big-O notation for this operation will be: $O(n)$.
We can optimize this by changing our anagrams method for a less complex one.

2.1.1 Optimizing the anagrams comparison

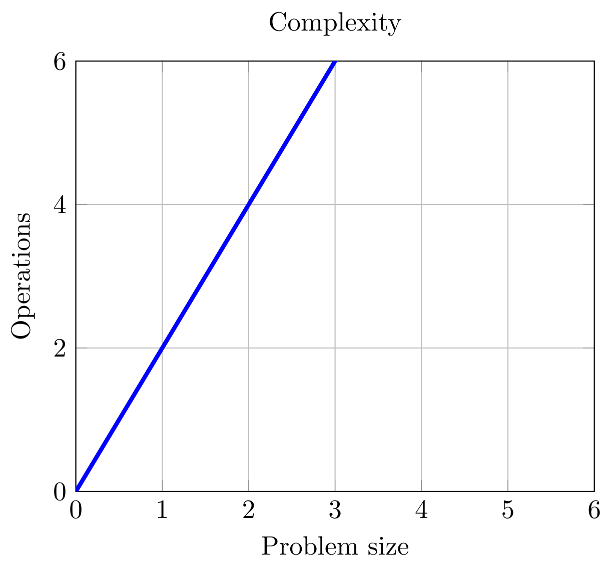
The new challenge is to optimize the method that determines if two words are anagrams or not.
As we can see, we have the same length and the same letters in words for them to be anagrams. This gives us a clue for a criteria to determine whether they are anagrams or not.
If we have the same characters in both Strings, we will have the same String if we sort both Strings. Then, we can compare them. If we have the same, then those Strings are anagrams. Now the issue is that comparing two Strings means n operations, where n is the length of the Strings. This is not a big issue as we finally have a $T(n)_2 < 2n$:

$$T(n)_2 = \underbrace{log(n)}_{\text{from quicksorting String x}} + \underbrace{log(n)}_{\text{from quicksorting String y}} + \underbrace{n}_{\text{from comparison}}$$

$$T(n)_2 = 2(log(n)) + n$$

Now, let's compare $T(n)_1$ and $T(n)_2$:

$T(n)_1$
$T(n)_2$



As we can see in the graph above, the $T(n)_2$ *red* curve grows slower than $T(n)_1$ *blue* curve.

Conclusion: The best way to determine if some words are anagrams is by sorting and comparing them.

2.1.2 Auxiliary dictionary

In order to keep the original dictionary, we create an auxiliary dictionary which will contain the same words but with their characters sorted with **quicksort**. First of all, we have to copy the original dictionary to the auxiliary one, which will take n operations, where n is the number of elements in the dictionary.

$$T(n)_3 = \underbrace{n}_{\text{from copying the original dictionary to the auxiliary}}$$

Then, we have to sort the characters in the words in the auxiliary dictionary. To achieve this, we will have to to to every element in the list, which will take n operations, where n is the number of elements in the auxiliary dictionary. The **quicksort** will take $log(n)$ steps.

$$T(n)_4 = \underbrace{n}_{\text{from the number of elements}} \times \underbrace{log(n)}_{\text{from }\textbf{quicksort}}$$

2.2 Hola