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:

"neotowheret"

Matches with the first word in the dictionary:

{"one", "two", "three", "there"}

Then we have the second visible valid word:

"neotowheret"

It matches with the second word in the dictionary:

{"one", "two", "three", "there"}

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

{"one", "two", "three", "there"}

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

"neotowheret"

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” ≠ “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” ≠ “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” ≠ “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:

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

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: $word \times word \rightarrow \{true, false\}$ 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 Determining 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:

$$T(m)_1 = \underbrace{mk}_{x} + \underbrace{mk}_{y}$$

$$O(m^2)$$

Where m 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: $O(m^2)$.
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 m operations, where m is the length of the Strings. This is not a big issue as we finally have a $T(m)_2 < 2m$:

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

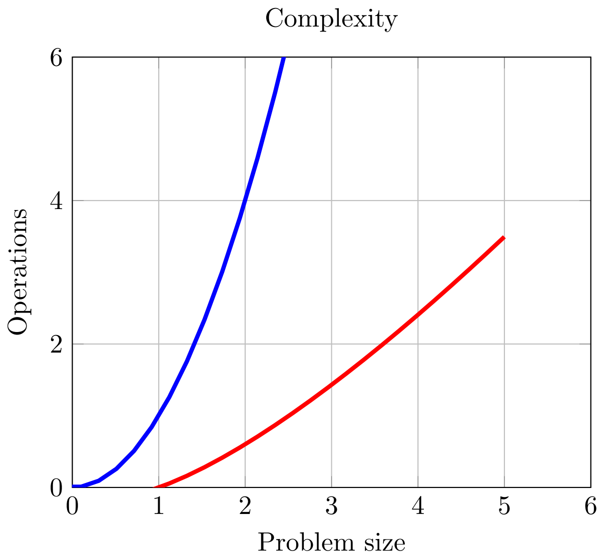
$$T(m)_2 = 2(m \times log(m)) + m$$

$$O(m \times log(m))$$

Where m is the length of the String.

Now, let's compare $O(m^2)$ and $O(m \times log(m))$:

$$O(m^2)$$
$$O(m \times log(m))$$



As we can see in the graph above, the $O(m \times log(m))$ curve grows slower than $O(m^2)$ curve.

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

2.2 Preparing for dynamic programming

We are going to begin our algorithm adapting it to work with **dynamic programming**.

2.2.1 Copying the original sentence length

We start creating a length variable to copy in it the length of the initial sentence. We do this because the sentence will be altered during the further algorithm operations. This will take a constant number of operations of 1.

$$T(m,n,p)_{\text{total}} = \underbrace{1}_{\text{from copying the length into a new variable}}$$

2.2.2 Creating dynamic programming array

In this part, we are concerned about creating an array for memorization of operations answers, in order to reduce the complexity by avoiding the operation repetition. This will take a constant number of operations of 1 that will be added to the total number of operations.

$$T(m,n,p)_{\text{total}} = 1 + \underbrace{1}_{\text{from creating the dynamic programming array}}$$

2.2.3 Filling dynamic programming array with max value

In order to identify the base cases and general cases, we need to fill our **dynamic programming** array with the value from **Integer.MAX_VALUE**. This will take $n + 1$ operations which will be added to the total number of operations.

$$T(m, n, p)_{\text{total}} = 1{+}1{+}\underbrace{n + 1}_{\text{from filling \textbf{dynamic programming} array with \textbf{Integer.MAX\_VALUE}}}$$

Where n is the length of the sentence to be analyzed. In this case, we have to add 1, which corresponds to the copied sentence length + 1.

Now, we are going to assign the value 0 to the first element in the **dynamic programming** array. This will take a constant number of operations of 1 that will be added to the total number of operations.

$$T(m, n, p)_{\text{total}} = 1{+}1{+}n{+}1{+}\underbrace{1}_{\text{from assigning 0 to the \textbf{dynamic programming} array in position 0}}$$

2.3 Finding anagrams

2.3.1 Determining iteration and general cases

Now, we are going to declare a loop which is going to iterate n times (remember that n is the sentence length). Nevertheless, this will not happen every time. We have to check first whether the position in the array in the current i iteration value is less than **Integer.MAX_VALUE**. As values inside the array will be changing, this verification will take 1 operation and as it will be done in every iteration.

$$T(m, n, p)_{\text{total}} = 1 + 1 + n + 1 + 1 + [\underbrace{1}_{\text{verification}} \times \underbrace{n}_{\text{iterations}}]$$

Then we have to do another conditional iteration inside the one above. This one will go through the words in the dictionary. This conditional iteration will happen only if **two conditions** are satisfied:

- The sum between the current i iteration and the length of the current word in the dictionary is equal or less than the length of the sentence being analyzed. This is necessary because the problem definition will no have sense if any word in the dictionary is longer than the sentence. That sentence couldn't be formed. This will take 1 operation for the addition and 1 for comparing with the *length*.

  $$T(m, n, p)_{\text{total}} = 1 + 1 + n + 1 + 1 + \{1 \times n \times [\underbrace{1}_{\text{addition}} + \underbrace{1}_{\text{comparison}}]\}$$

  Simplifying the expression we get:

  $$T(m, n, p)_{\text{total}} = 4 + n + \{n \times 2\}$$

- The current word in the dictionary and the subsequence being analyzed must be anagrams. If they are not, there wouldn't exist a transformation such that we can reach the target sequence with the characters in the current subsequence. This will take m for the subsequence extraction steps. This will be inside the method for determining if two words are anagrams, which takes $2(m \times log(m)) + m$ steps.

$$T(m,n,p)_{\text{total}} = 4+n+\{n\times 2\times[\underbrace{m}_{\text{subsequence}} \times \underbrace{(2(m \times log(m)) + m)}_{\text{anagrams}} \times \underbrace{p}_{\text{dictionary size}}]\}$$

2.4 Cost operations and minimization

Now that we have determined the constraints for our iterations, we must solve the most important part of our problem which is determining and minimizing the cost to transform from one String to another. The following operations will be described assuming that the iteration constraints have been satisfied.

What we have to do now is to memorize with **dynamic programming**. We are going to assign into the array positions, the minimum values of cost that solve our problem. This values are found with two constraints for each possible couple that can be minimized:

- In the current position, we start checking from left to right with the cost for transformation of the current subsequence into the dictionary word. This will take
- The latest infallible element plus the cost of the current word that is being analyzed. This will take $1 + m \times m$ operations.

$$\begin{aligned} T(m,n,p)_{\text{total}} = 4 + n + \{n \times 2 \times [m \times (2(m \times log(m)) \\ + m) \times p \times (\underbrace{1}_{\text{array access}} + \underbrace{m}_{\textbf{\textit{cost}}} \times \underbrace{m}_{\text{subsequence}})]\} \end{aligned}$$

We are going to pick the minimum between this two. This will take 1 operation from the comparison, plus the steps taken by both possibilities which have to be done to get the minimum
Finally, we return the value in the length position because this one is the last one calculated, which contains the total, that has been growing from the previous minimum ***cost*** values.

2.5 Final complexity

After solving the expression, we get a final asymptotic complexity of: $O(n \times m^2 \times log(m))$.