Programming Assignment 2 Analysis

EN.605.621
Foundations of Algorithms
Spring 2023

Jose Márquez Jaramillo

Engineering and Applied Science Programs for Professionals
Johns Hopkins University
United States
May 1st, 2023

Contents

1 The important concepts

By the problem definition, we know that we will be given three particular strings. s corresponds to a set of symbols, and we are testing to see if s is an interweaving string composed of two other sets of symbols x and y. The following information is given regarding the string s:

1. We do not know if the signal s received has its first symbol in x, y or a symbol that is extra and added in,
2. we know that s must be at least as long as the sum of the lengths of x and y,
3. and if we cannot determine that the signal is an interweaving of x and y, either because s is not long enough to do so or because s contains symbols not in x or y then we cannot decide that s is an interweaving of x and y

Let us consider the information provided above. All the elements of x and y need to be contained in s, this means that we need to check whether the elements can be found. There could be several ways to do so. While performing this check, instead of checking and re-checking past elements of s again (which would result in exponential time complexity), we can keep track of the elements of x and y we have already found. This particular characteristic of the problem solution leads the way to the memoization property of dynamic programming.

According to section 6.2 of [Kleinberg and Tardos, 2006] about developing an algorithm based on dynamic programming, one needs a collection of sub-problems derived from the original problem that satisfies a few basic properties:

1. **There are only a polynomial number of sub-problems:** we know that checking whether a particular symbol belongs to x or y can be achieved with a nested loop over them, therefore can easily be achieved through quadratic time complexity.
2. **The solution to the original problem can be easily computed from the solutions to the sub-problems:** As we move through the string s and store the result up to the previous symbol, we know whether we are ”ok” to continue or not. That is, past solutions indicate whether the next solution is feasible.
3. **There is a natural ordering on sub-problems from “smallest” to “largest,” together with an easy-to-compute recurrence:** Since the English language is written and read from left to right, this would be the natural way of solving the problem, although the accumulation can happen also from right to left without any significant impact.

2 An illustrative example

During our class office hours for Module 11, Dr. Leon presented a couple of examples. I think it is of benefit to illustrate a mechanism to check s to further explain the implementation of the algorithm.

Let $x = 101$ and $y = 00100$. From the previous statements we know that for s to be an interweave of x and y:

1. s should have at least $3 + 5 = 8$ elements;
2. s should contain at least one full x and one full y;

3. s can have leading or trailing symbols;
4. s cannot have intermediate symbols that do not belong to x or y.

We can present a table that would indicate feasible combinations of x and y, similar to the tabular representation of a dynamic program

x/y	0	0	1	0	0
1					
0					
1					

Table 1: Feasible example options

Let's provide a few examples (assume reading them left to right):

1. $s = 100000100$: is not an interweave because there are 4 consecutive 0s, which neither x or y have.
2. $s = 10001001$: is an interweave because the positions $1, 2,$ and 8 create x while the rest create y.
3. $s = 100011001$: is an interweave with a trailing 1 at the end.
4. $s = 1000111001$: is not an interweave since there is an additional 1 in between the reproductions of x and y.

From this illustration, it should be clear that the objective should be to fill a table that checks if s contains a sub-string that has possible combinations of x and y. Keeping track in a table such as Table 1 should alleviate the computational complexity greatly.

3 A dynamic program

Algorithm 1 HelperFunction(x,y,s)

Inputs: Strings x,y and s, **Output:** A boolean value

```
1: $n \leftarrow$ size of $x$
2: $m \leftarrow$ size of $y$
3: $DP \leftarrow$ array of False values with dimensions $(m+1) \times (n+1)$
4: $DP[0][0] =$ True
5: for $i \in [0 : n+1]$ do
6:     $DP[i][0] = DP[i-1][0] \& (x[i-1] == s[i-1])$
7: end for
8: for $i \in [0 : m+1]$ do
9:     $DP[0][i] = DP[i-1][0] \& (y[i-1] == s[i-1])$
10: end for
11: for $i \in [1 : n+1]$ do
12:     for $j[1 : m+1]$ do
13:         if $x[i-1] == s[i+j-1]$ then
14:             $DP[i][j] = DP[i-1][j]$
15:         end if
16:         if $y[j-1] == s[i+j-1]$ then
17:             $DP[i][j] = DP[i-1][j]$ or $DP[i][j-1]$
18:         end if
19:     end for
20: end for
21: return $DP[-1][-1]$
```

Algorithm 2 IsInterweave(x,y,s)

Inputs: Strings x,y and s, **Output:** A boolean value

```
1: $i \leftarrow 0$
2: while length($s$) $\geq$ length($x$) + length($x$) do
3:     $v \leftarrow$ HelperFunction($x, y, s[i:]$)
4:     if $v$ then
5:         return $v$
6:     end if
7:     $i = i + 1$
8: end while
9: return False
```

Algorithm 3 RecIsInterweave(x,y,s)

Inputs: Strings x,y and s, **Output:** A boolean value

```
    $i \leftarrow 0$
    if length($s$)<length($x$)+length($y$) then
3:      return False
    end if
    $v \leftarrow$ HelperFunction($x, y, s[i:]$)
6:  if $v$ then
        return True
    else
9:      return RecIsInterweave($x$,$y$,$s[i+1:]$)
    end if
    return False
```

The program is composed of two particular components: Algorithm 1 corresponds to a dynamic program that uses the table mechanism explored in the previous section, Algorithm 2 presents a loop to parse through the string, and Algorithm 3 uses recursion to achieve the same. Below I will go through the algorithms by line to better explain:

3.1 HelperFunction [1]

Lines 1 and 2 of HelperFunction assign the sizes of x and y to n and m respectively. Line 3 creates a two-dimensional array, acting effectively as the example presented in Table 1. It is important to note that the dimensions of the table are $(n+1) \times (m+1)$, this is because we need to accommodate for an initialization place at position $[0][0]$. This initialization occurs at line 5 with a Boolean True value. Next, lines 5 through 10 initialize True values for the first row corresponding to the initialization row on x and the first column on y. So far we can see that we have linear running time. The loop in lines $5-7$ runs at $O(n+1)$ and the loop in lines $8-10$ runs at $O(m+1)$.

Next, the dynamic program takes place in lines 11 through 20. It is important to note first that the dynamic program happens in a nested loop. This indicates that as we scan through the dynamic table, the time complexity is $O((n+1) \times (m+1)) = O(nm)$. I will now discuss briefly what happens in between:

- Lines $13-15$ check if the previous position in $x[i]$ matches the position $s[i+j-1]$, effectively scanning through the x string to check if I have matched a previous valid position in the table. If so, the dynamic table in position $[i][j]$ gets updated to the previous value corresponding to that position coming from the x string.

- Lines $16-18$ do about the same but using the string y instead. Notice that in line 17 there is a slight difference in line 17. We have an or statement that will update the current position $[i][j]$ in the dynamic table if the previous positional row or column of the table has a True value.

The nested for loop in $11-20$ navigates the Dynamic Table effectively. Finally, as we navigated the table from the top left, we return the value at the bottom right, or position $[-1][-1]$. If the value at position $[-1][-1]$ is True, we have a string s that effectively interleaves x and y.

Notice however that we still have a significant shortcoming, if the string s has preceding symbols which are not contained in x or y, the table will not update into the feasible dynamic table. On the other hand, since the dynamic table is filled out regardless of the length of s. If s is found to be an interleaving string, all the succeeding symbols will not affect the value of the dynamic table at $[-1][-1]$ if an appropriate feasible string is traversed. This property takes care of succeeding symbols, the next function takes care of preceding symbols.

3.2 IsInterweave [2] and RecIsInterweave [3]

I would like to first note that both IsInterweave [2] and RecIsInterweave [3] accomplish the exact same task. RecIsInterweave [3] is simply a recursive version of IsInterweave [2]. The reason for having both is that as I scaled the string length the recursive function started to cause performance issues on my machine.

I previously stated that HelperFunction [1] takes care of the dependency on succeeding symbols, yet it does not for preceding symbols. IsInterweave [2] scans through the string changing the initial one place at a time. If while scanning through the string it is found that the string is interleaving with preceding symbols, then True is returned, and the loop breaks. We also have the condition that the length of s cannot be shorter than the sum of the lengths of x and y, therefore we account for that using it as the condition in the while loop. By inspecting the loop, we can see that IsInterweave [2] runs linearly as a function of the length of s, in other words, it is $O(\text{length}(s))$.

4 Algorithm time complexity

I have previously stated that individually HelperFunction [1] runs at $O(mn)$ while IsInterweave [2] runs at $O(\text{length}(s))$. Since IsInterweave [2] is a wrapper function of HelperFunction [1], we can expect the total algorithm to run a total of $O(\text{length}(s)) \times O(mn)$ or at $O(\text{length}(s) \times mn)$. This is certainly polynomial and I will contend next through graphical depictions that the running time is in fact linear.

In order to account for the running time, the assignment indicates to use of the number of comparisons as a valid time complexity counter. Therefore, in the actual algorithm implementation, I have incorporated a variable $comp$ which is used to accumulate the total number of comparisons in a particular run. Within the Analysis directory of the submitted folder, you will find a jupyter notebook called Analysis.jpynb which has the code that produces the graphs shown below for analysis.

4.1 Testing an interweaved string s

Consider the example when $x = 101, y = 00100$, and $s = 100011001$. You may recall that this is an example 3 in section 2. We know that this s is an interweaving string of x and y. In order to increase the sample size or length of s, I will concatenate zeroes at the beginning of the string, effectively making the algorithm scan for greater lengths as the string s scales.

The figure below shows the results of increasing the size of s up to $20,000$ symbols while keeping $s =' 100011001'$ at the end. Consider that this method an approximation of the worst-case scenario, this is due to the fact that we are moving the actual string to be tested further to the right on every iteration. This method then serves to ratify that the algorithm indeed runs at $O(\text{length}(s) \times mn)$ as shown in Figure 1.

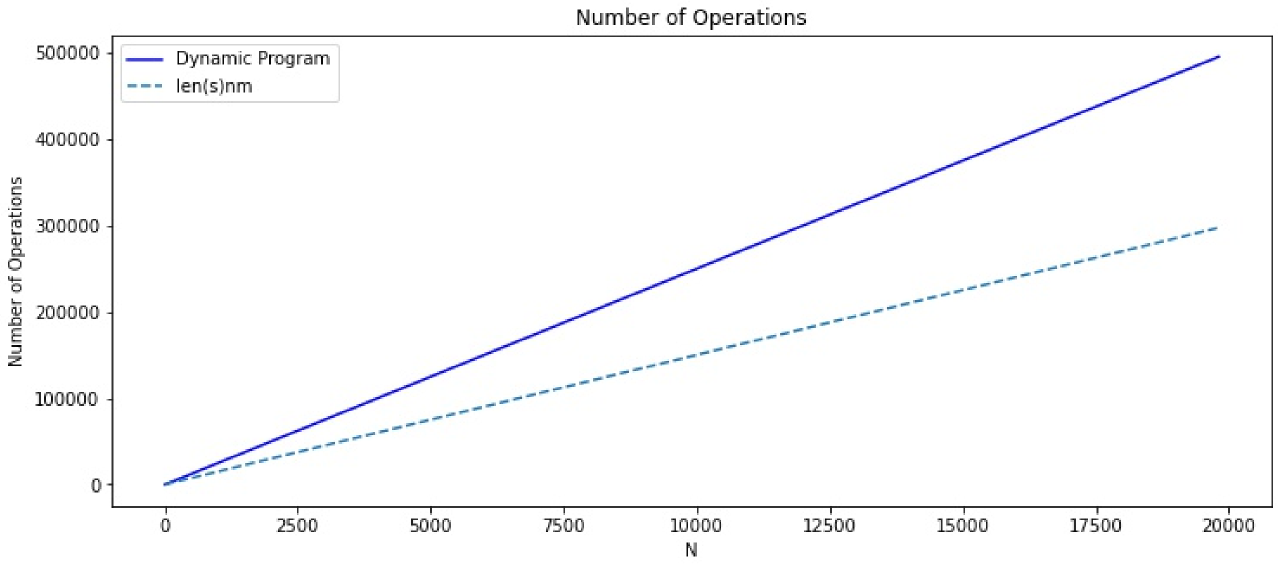

Figure 1: Moving the interweave string forward

We can easily see from Figure 1 that although the number of comparisons exceeds the actual value of the function length$(s) \times mn$, we could easily find a constant value c such that $f(x) \leq c \times$ length$(s) \times mn$, confirming that $f(x) = O($length$(s) \times mn)$.

4.2 Testing random strings

Another procedure I have implemented to test for time complexity involves generating random strings for x, y, and s. That is, not only are the strings themselves random, but the lengths of x and y are also random from $0 - 15$. Using a random binary generator we can produce random strings of 1s and 0s consistently and test whether the random string s is an interweaving of the random strings x and y. Keep in mind that because of the randomness of the inputs, the value of length$(s) \times mn$ is not linear and does not exhibit a recognizable pattern. What we want to test however is whether the number of comparisons in the algorithm is bounded by the function length$(s) \times mn$.

Figure 2 below confirms a similar conclusion to the one through Figure 1. It is interesting to see that although we have added a significant level of randomness, the function algorithm behaves as expected.

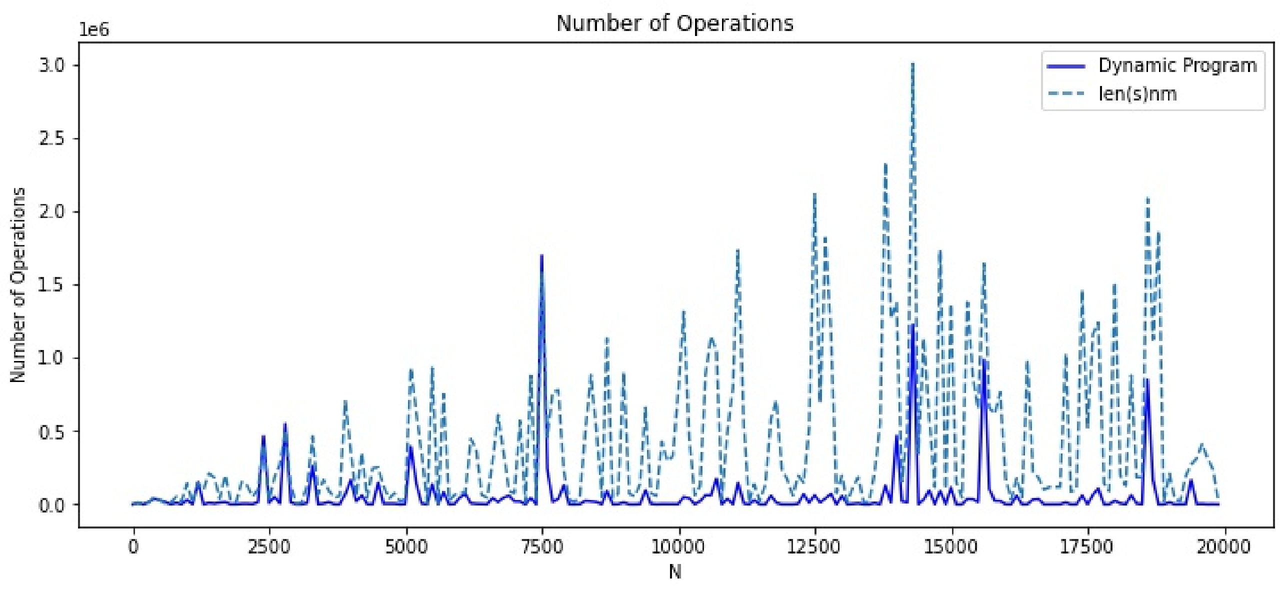

Figure 2: Using random strings

4.3 Retrospection

Even though I believe that this program is can be rather trivial for a seasoned dynamic programmer, it proved quite challenging for me. Although in lectures we reviewed and saw several examples of dynamic programs, the actual development of a program from scratch proved quite difficult. However, it was very helpful to achieve a much better understanding of the mechanics of dynamic programs in general.

I am sure that once I become more avid at dynamic programming I could probably find better ways to achieve the result, the one thing that really jumps out at the moment and which I believe can be improved is having to traverse the string and call the dynamic program at each iteration. There is probably a better way to approach the problem dynamically within what I currently call the HelperFunction [1].

References

[Kleinberg and Tardos, 2006] Kleinberg, J. and Tardos, E. (2006). *Algorithm Design*. Addison Wesley.