Given two strings `text1` and `text2`, return *the length of their longest* ***common subsequence***. If there is no **common subsequence**, return `0`.

A **subsequence** of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters.

- For example, `"ace"` is a subsequence of `"abcde"`.

A **common subsequence** of two strings is a subsequence that is common to both strings.

Example 1:

```
Input: text1 = "abcde", text2 = "ace"
Output: 3
Explanation: The longest common subsequence is "ace" and its length is 3.
```

Example 2:

```
Input: text1 = "abc", text2 = "abc"
Output: 3
Explanation: The longest common subsequence is "abc" and its length is 3.
```

Example 3:

```
Input: text1 = "abc", text2 = "def"
Output: 0
Explanation: There is no such common subsequence, so the result is 0.
```

Constraints:

- `1 <= text1.length, text2.length <= 1000`
- `text1` and `text2` consist of only lowercase English characters.

Hide Hint #1

Try dynamic programming. DP[i][j] represents the longest common subsequence of text1[0 ... i] & text2[0 ... j].

Hide Hint #2

DP[i][j] = DP[i - 1][j - 1] + 1 , if text1[i] == text2[j] DP[i][j] = max(DP[i - 1][j], DP[i][j - 1]) , otherwise

Solution

Overview

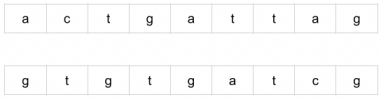
This is a nice problem, as unlike some interview questions, this one is a real-world problem! Finding the longest common subsequence between two strings is useful for checking the difference between two files (diffing). Git needs to do this when merging branches. It's also used in genetic analysis (combined with other algorithms) as a measure of similarity between two genetic codes.

For that reason, the examples used in thi leetcode_com.pdf of the letters `a`, `c`, `g`, and `t`. You might remember these letters from high school biology—they are the symbols we use to represent genetic codes. By using just four letters in examples, it is easier for us to construct interesting examples to discuss here. You don't need to know *anything* about genetics or biology for this though, so don't worry.

Before we look at approaches that *do* work, we'll have a quick look at some that do not. This is because we're going to pretend that you've just encountered this problem in an interview, and have never seen it before, and have *not* been told that it is a "dynamic programming problem". After all, in this interview scenario, most people won't realize immediately that this is a dynamic programming problem. Being able to approach and explore problems with an open mind without jumping to early conclusions is essential in tackling problems you haven't seen before.

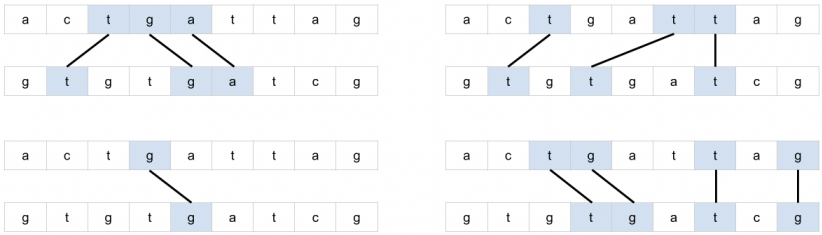
What is a Common Subsequence?

Here's an example of two strings that we need to find the longest common subsequence of.

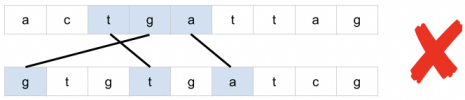


A common subsequence is a sequence of letters that appears in both strings. Not every letter in the strings has to be used, but letters cannot be rearranged. In essence, a subsequence of a string `s` is a string we get by deleting some letters in `s`.

Here are some of the common subsequences for the above example. To help show that the subsequence really is a common subsequence, we've drawn lines between the corresponding characters.



Drawing lines between corresponding letters is a great way of visualizing the problem and is potentially a valuable technique to use on a whiteboard during an interview. Observe that if lines cross over each other, then they do *not* represent a common subsequence.



This is because lines that cross over are representing letters that have been rearranged.

We will use and refer to "lines" between the words extensively throughout this article.

Brute-force

The most obvious approach would be to iterate through each subsequence of the first string and check whether or not it is also a subsequence of the second string.

This, however, will require exponential time to run. The number of subsequences in a string is up to 2^L, where L is the length of the string. This is because, for each character, we have two choices; it can either be in the subsequence or not in it. Duplicates characters reduce the number of unique subsequences a bit, although in the general case, it's still exponential.

This would be a brute-force approach.

Greedy

By this point, it's hopefully clear that we're dealing with an *optimization problem*. We need to generate a *common* subsequence that *has the maximum possible number of letters*. Using our analogy of drawing lines between the words, we could also phrase it as *maximizing the number of non-crossing lines*.

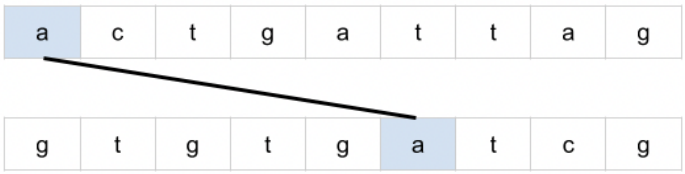
There are a couple of strategies we use to design a tractable (non-exponential) algorithm for an optimization problem.

1. Identifying a greedy algorithm
2. Dynamic programming

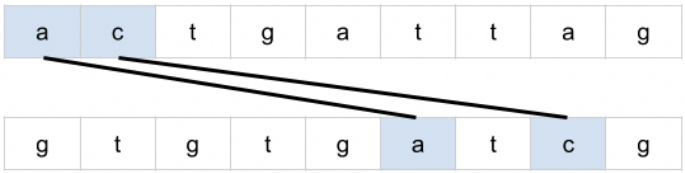
There is no guarantee that either is possible. Additionally, greedy algorithms are strictly less common than dynamic programming algorithms and are often more difficult to identify. However, *if* a greedy algorithm exists, then it will almost always be better than a dynamic programming one. You should, therefore, at least give some thought to the potential existence of a greedy algorithm before jumping straight into dynamic programming.

The best way of doing this is by drawing an example and playing around with it. One idea could be to iterate through the letters in the first word, checking whether or not it is possible to draw a line from it to the second word (without crossing lines). If it is, then draw the left-most line possible.

For example, here's what we would do with the first letter of our example from earlier.

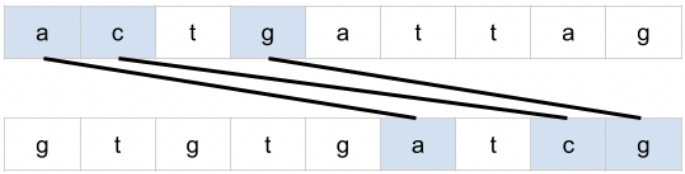


And then, the second letter.

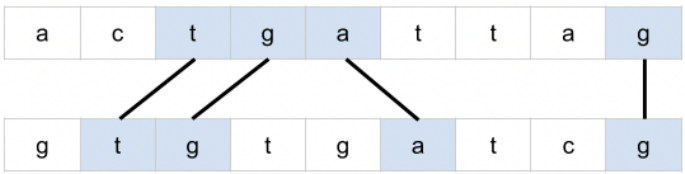


leetcode_com.pdf

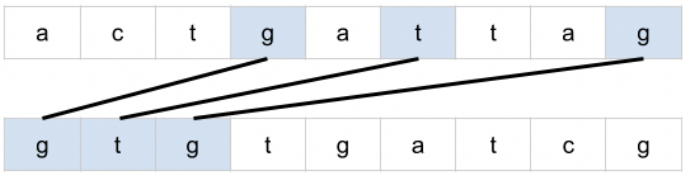
And finally, the third letter.



This solution, however, isn't optimal. Here is a better solution.

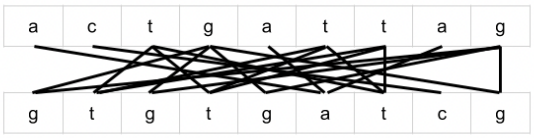


What if we were to do the same, but instead going from the second word to the first word? Perhaps one way or the other will always be optimal?



Unfortunately, this hasn't worked either. This solution is still worse than a better one we know about.

Perhaps, instead, we could draw all possible lines. Could there be a way of eliminating some of the lines that cross over?



Uhoh, we now have what looks like an even more complicated problem than the one we began with. With some lines crossing over many other lines, where would you even begin?

Applying Dynamic Programming to a Problem

While it's *very* difficult to be *certain* that there is no greedy algorithm for your interview problem, over time you'll build up an intuition about when to give up. You also don't want to risk spending so long trying to find a greedy algorithm that you run out of time to write a dynamic programming one (and it's also best to make sure you write a working solution!).

Besides, sometimes the process used to develop a dynamic programming solution can lead to a greedy one. So, you might end up being able to further optimize your dynamic programming solution anyway.

Recall that there are two different techniques we can use to implement a dynamic programming solution; memoization and tabulation.

- **Memoization** is where we add caching to a function (that has no side effects). In dynamic programming, it is typically used on **recursive** functions for a **top-down** solution that starts with the initial problem and then recursively calls itself to solve smaller problems.
- **Tabulation** uses a table to keep track of subproblem results and works in a **bottom-up** manner: solving the smallest subproblems before the large ones, in an **iterative** manner. Often, people use the words "tabulation" and "dynamic programming" interchangeably.

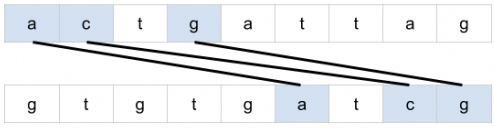
For most people, it's easiest to start by coming up with a recursive brute-force solution and then adding memoization to it. After that, they then figure out how to convert it into an (often more desired) bottom-up tabulated algorithm.

Approach 1: Memoization

Intuition

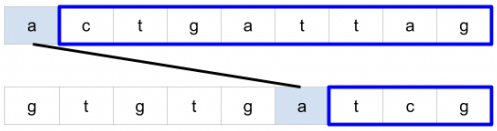
The first step is to find a way to recursively break the original problem down into subproblems. We want to find subproblems such that we can create an optimal solution from the results of those subproblems.

Earlier, we were drawing lines between identical letters.



Consider the greedy algorithm we tried earlier where we took the first possible line. Instead of assuming that the line is part of the *optimal solution*, we could consider both cases: *the line **is** part of the optimal solution* or *the line **is not** part of the optimal solution*.

If *the line **is** part of the optimal solution*, then we know that the rest of the lines must be in the substrings that follow the line. As such, we should find the solution for the substrings, and add `1` onto the result (for the new line) to get the *optimal solution*.



However, if *the line **is not** part of the optimal solution*, then we know that the letter in the first string is not included (as this would have been the best possible line for that letter). So, instead, we remove the first letter of the first string and treat the remainder as the subproblem. Its solution will be the *optimal solution*.

But remember, we don't know which of these two cases is true. As such, we need to compute the answer for **both** cases. The highest one will be the *optimal solution* and should be returned as the answer for this problem.

Note that if either the first string or the second string is of length 0, we don't need to break it into subproblems and can just return `0`. This acts as the base case for the recursion.

But how many total subproblems will we need to solve? Well, because we always take a character off one, or both, of the strings each time, there are $M \cdot N$ possible subproblems (where M is the length of the first string, and N the length of the second string). Another way of seeing this is that subproblems are represented as *suffixes* of the original strings. A string of length K has K unique suffixes. Therefore, the first string has M suffixes, and the second string has N suffixes. There are, therefore, $M \cdot N$ possible pairs of suffixes.

Some subproblems may be visited multiple times, for example `LCS("aac", "adf")` has the two subproblems `LCS("ac", "df")` and `LCS("ac", "adf")`. Both of these share a common subproblem of `LCS("c", "df")`. As such, as we should memoize the results of `LCS` calls so that the answers of previously computed subproblems can immediately be returned without the need for re-computation.

Algorithm

From what we've explored in the intuition section, we can create a top-down recursive algorithm that looks like this in pseudocode:

```
define function LCS(text1, text2):
    # If either string is empty there, can be no common subsequence.
    if length of text1 or text2 is 0:
        return 0

    letter1 = the first letter in text1
    firstOccurence = first position of letter1 in text2

    # The case where the line *is not* part of the optimal solution
    case1 = LCS(text1.substring(1), text2)

    # The case where the line *is* part of the optimal solution
    case2 = 1 + LCS(text1.substring(1), text2.substring(firstOccurence + 1))

    return maximum of case1 and case2
```

You might notice from the pseudocode that there's one case we haven't handled: if `letter1` isn't part of `text2`, then we can't solve the first subproblem. However, in this case, we can simply ignore the first subproblem as the line doesn't exist. This leaves us with:

```
define function LCS(text1, text2):
    # If either string is empty there can be no common subsequence
    if length of text1 or text2 is 0:
        return 0

    letter1 = the first letter in text1

    # The case where the line *is not* part of the optimal solution
    case1 = LCS(text1.substring(1), text2)

    case2 = 0
    if letter1 is in text2:
        firstOccurence = first position of letter1 in text2
        # The case where the line *is* part of the optimal solution
        case2 = 1 + LCS(text1.substring(1), text2.substring(firstOccurence + 1))

    return maximum of case1 and case2
```

Remember, we need to make sure that the results of this method are memoized. In **Python**, we can use `lru_cache`. In **Java**, we need to make our own data structure. A 2D Array is the best option (see the code for the details of how this works).

Java | Python3 — Copy

```java
  }

  private int memoSolve(int p1, int p2) {
    // Check whether or not we've already solved this subproblem.
    // This also covers the base cases where p1 == text1.length
    // or p2 == text2.length.
    if (memo[p1][p2] != -1) {
      return memo[p1][p2];
    }

    // Option 1: we don't include text1[p1] in the solution.
    int option1 = memoSolve(p1 + 1, p2);

    // Option 2: We include text1[p1] in the solution, as long as
    // a match for it in text2 at or after p2 exists.
    int firstOccurence = text2.indexOf(text1.charAt(p1), p2);
    int option2 = 0;
    if (firstOccurence != -1) {
      option2 = 1 + memoSolve(p1 + 1, firstOccurence + 1);
    }

    // Add the best answer to the memo before returning it.
    memo[p1][p2] = Math.max(option1, option2);
    return memo[p1][p2];
  }
}
```

Complexity Analysis

- Time complexity : $O(M \cdot N^2)$.

  We analyze a memoized-recursive function by looking at how many unique subproblems it will solve, and then what the cost of solving each subproblem is.

  The input parameters to the recursive function are a pair of integers; representing a position in each string. There are M possible positions for the first string, and N for the second string. Therefore, this gives us $M \cdot N$ possible pairs of integers, and is the number of subproblems to be solved.

  Solving each subproblem requires, in the worst case, an $O(N)$ operation; searching for a character in a string of length N. This gives us a total of $(M \cdot N^2)$.

- Space complexity : $O(M \cdot N)$.

  We need to store the answer for each of the $M \cdot N$ subproblems. Each subproblem takes $O(1)$ space to store. This gives us a total of $O(M \cdot N)$.

It is important to note that the time complexity given here is an *upper bound*. In practice, many of the subproblems are *unreachable*, and therefore not solved.

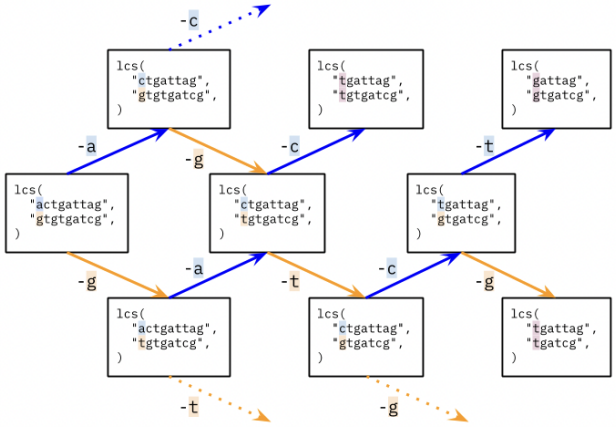
For example, if the first letter of the first string is not in the second string, then only one subproblem that has the entire first word is even considered (as opposed to the N possible subproblems that have it). This is because when we search for the letter, we skip indices until we find the letter, skipping over a subproblem at each iteration. In the case of the letter not being present, no further subproblems are even solved with that particular first string.

Approach 2: Improved Memoization

Intuition

There is an alternative way of expressing the solution recursively. The code is simpler, and will also translate a lot more easily into a bottom-up dynamic programming approach.

The subproblems are of the same structure as before; represented as two indexes. Also, like before, we're going to be considering multiple possible decisions and then going with the one that has the highest answer. The difference is that the way we break a problem into subproblems is a bit different. For example, here is how our example from before breaks into subproblems.

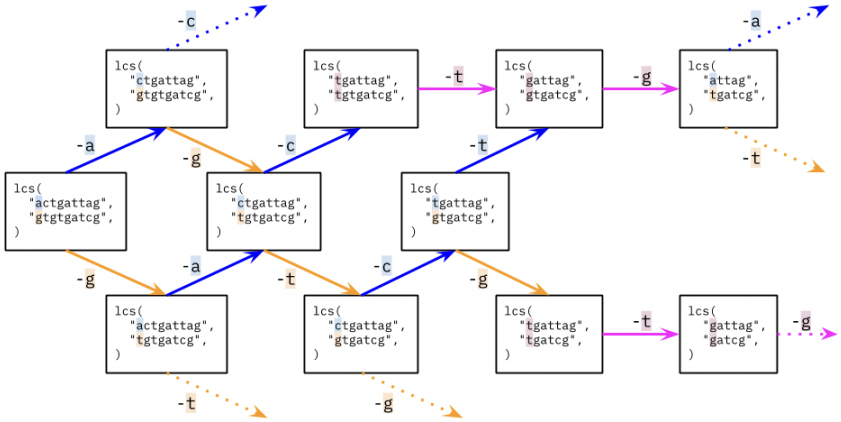


> If the first character of each string is **not** the same, then either *one or both* of those characters will not be used in the final result (i.e. not have a line drawn to or from it). Therefore, the length of the longest common subsequence is `max(LCS(p1 + 1, p2), LCS(p1, p2 + 1))`.

Now, what about subproblems such as `LCS("tgattag", "tgtgatcg")`? The first letter of each string is the same, and so we could draw a line between them. Should we? Well, there is no reason not to draw a line between the *first* characters when they're the same. This is because it won't block any later (optimal) decisions. No letters other than those used for the line are removed from consideration by it. Therefore, we don't need to make a decision in this case.

> When the first character of each string is the same, the length of the longest common subsequence is `1 + LCS(p1 + 1, p2 + 1)`. In other words, we draw a line a line between the first two characters, adding `1` to the length to represent that line, and then solving the resulting subproblem (that has the first character removed from each string).

Here is a few more of the subproblems for the above example.



Like before, we still have overlapping subproblems, i.e. subproblems that appear on more than one branch. Therefore, we should still be using a memoization table, just like before.

Algorithm

Java | Python3 — Copy

```java
class Solution {

  private int[][] memo;
  private String text1;
  private String text2;

  public int longestCommonSubsequence(String text1, String text2) {
    // Make the memo big enough to hold the cases where the pointers
    // go over the edges of the strings.
    this.memo = new int[text1.length() + 1][text2.length() + 1];
    // We need to initialise the memo array to -1's so that we know
    // whether or not a value has been filled in. Keep the base cases
    // as 0's to simplify the later code a bit.
    for (int i = 0; i < text1.length(); i++) {
      for (int j = 0; j < text2.length(); j++) {
        this.memo[i][j] = -1;
      }
    }
    this.text1 = text1;
    this.text2 = text2;
    return memoSolve(0, 0);
  }

  private int memoSolve(int p1, int p2) {
    // Check whether or not we've already solved this subproblem.
    // This also covers the base cases where p1 == text1.length
```

```
class Solution {

  private int[][] memo;
  private String text1;
  private String text2;

  public int longestCommonSubsequence(String text1, String text2) {
    // Make the memo big enough to hold the cases where the pointers
    // go over the edges of the strings.
    this.memo = new int[text1.length() + 1][text2.length() + 1];
    // We need to initialise the memo array to -1's so that we know
    // whether or not a value has been filled in. Keep the base cases
    // as 0's to simplify the later code a bit.
    for (int i = 0; i < text1.length(); i++) {
      for (int j = 0; j < text2.length(); j++) {
        this.memo[i][j] = -1;
      }
    }
    this.text1 = text1;
    this.text2 = text2;
    return memoSolve(0, 0);
  }

  private int memoSolve(int p1, int p2) {
    // Check whether or not we've already solved this subproblem.
    // This also covers the base cases where p1 == text1.length
    // or p2 == text2.length.
    if (memo[p1][p2] != -1) {
      return memo[p1][p2];
    }

    // Recursive cases.
    int answer = 0;
    if (text1.charAt(p1) == text2.charAt(p2)) {
      answer = 1 + memoSolve(p1 + 1, p2 + 1);
    } else {
      answer = Math.max(memoSolve(p1, p2 + 1), memoSolve(p1 + 1, p2));
    }

    // Add the best answer to the memo before returning it.
    memo[p1][p2] = answer;
    return memo[p1][p2];
  }
}
```

Complexity Analysis

- Time complexity : $O(M \cdot N)$.

  This time, solving each subproblem has a cost of $O(1)$. Again, there are $M \cdot N$ subproblems, and so we get a total time complexity of $O(M \cdot N)$.

- Space complexity : $O(M \cdot N)$.

  We need to store the answer for each of the $M \cdot N$ subproblems.

Approach 3: Dynamic Programming

Intuition

In many programming languages, iteration is faster than recursion. Therefore, we often want to convert a top-down memoization approach into a bottom-up dynamic programming one (some people go directly to bottom-up, but most people find it easier to come up with a recursive top-down approach first and then convert it; either way is fine).

> Observe that the subproblems have a natural "size" ordering; the largest subproblem is the one we start with, and the smallest subproblems are the ones with just one letter left in each word. The answer for each subproblem depends on the answers to some of the smaller subproblems.

Remembering too that each subproblem is represented as a pair of indexes, and that there are `text1.length() * text2.length()` such possible subproblems, we can iterate through the subproblems, starting from the smallest ones, and storing the answer for each. When we get to the larger subproblems, the smaller ones that they depend on will already have been solved. The best way to do this is to use a 2D array.

	g	t	g	t	g	a	t	c	g
a									
c									
t									
g									
a									
t									
t									
a									
g									

Each cell represents one subproblem. For example, the below cell represents the subproblem `lcs("attag", "gtgatcg")`.

Complexity Analysis

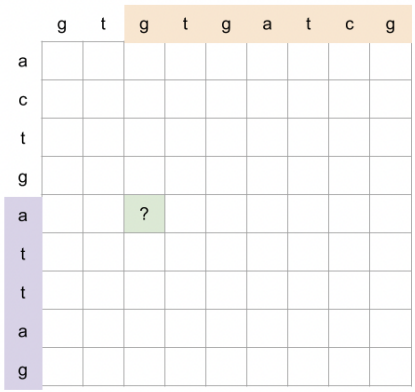
- Time complexity : $O(M \cdot N)$.

  This time, solving each subproblem has a cost of $O(1)$. Again, there are $M \cdot N$ subproblems, and so we get a total time complexity of $O(M \cdot N)$.

- Space complexity : $O(M \cdot N)$.

  We need to store the answer for each of the $M \cdot N$ subproblems.

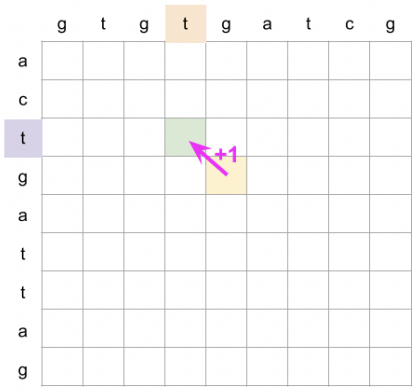
Each cell represents one subproblem. For example, the below cell represents the subproblem `lcs("attag", "gtgatcg")`.



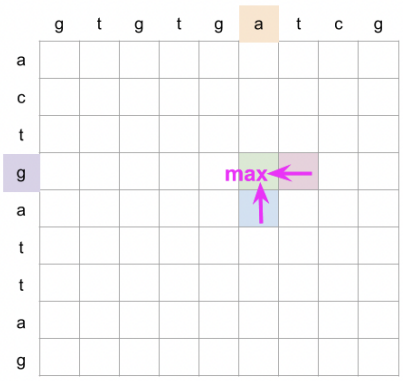
Remembering back to Approach 2, there were two cases.

1. The first letter of each string is the same.
2. The first letter of each string is different.

For the first case, we solve the subproblem that removes the first letter from each, and add `1`. In the grid, this subproblem is always the diagonal immediately down and right.



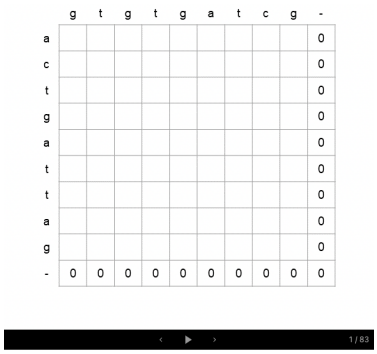
For the second case, we consider the subproblem that removes the first letter off the first word, and then the subproblem that removes the first letter off the second word. In the grid, these are subproblems immediately right and below.

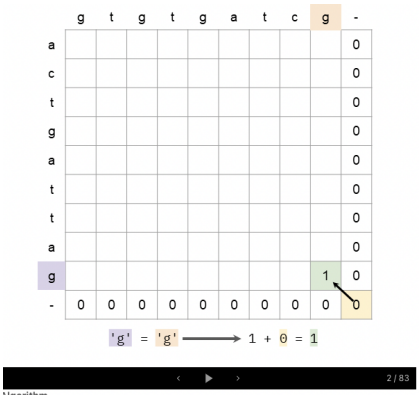


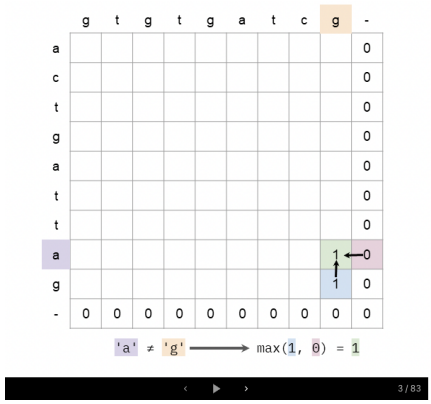
Putting this all together, we iterate over each column in reverse, starting from the last column (we could also do rows, the final result will be the same). For a cell `(row, col)`, we look at whether or not `text1.charAt(row) == text2.charAt(col)` is true. *if it is*, then we set `grid[row][col] = 1 + grid[row + 1][col + 1]`. Otherwise, we set `grid[row][col] = max(grid[row + 1][col], grid[row][col + 1])`.

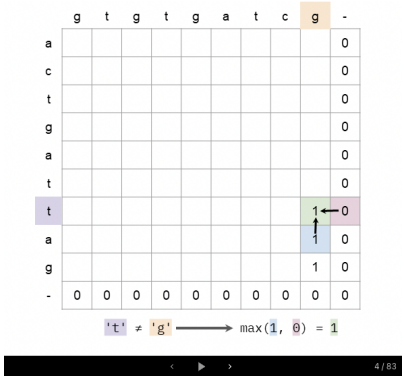
For ease of implementation, we add an extra row of zeroes at the bottom, and an extra column of zeroes to the right.

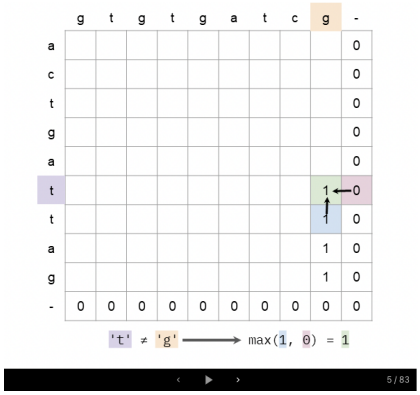
Here is an animation showing this algorithm.

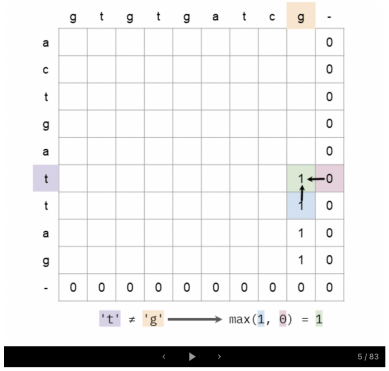


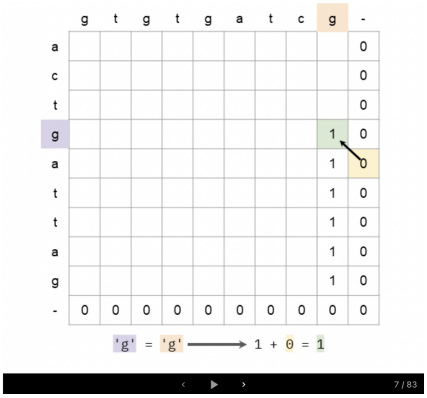


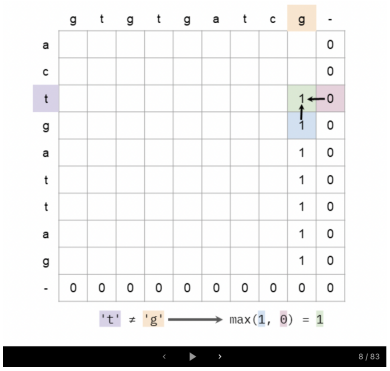


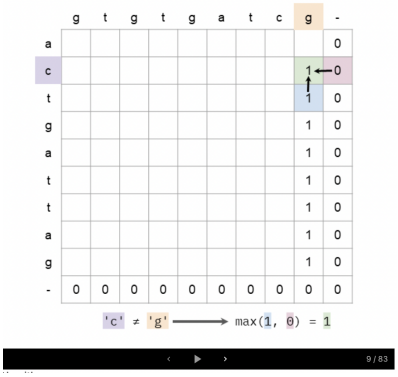


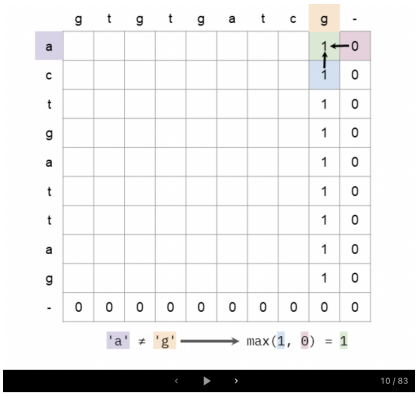


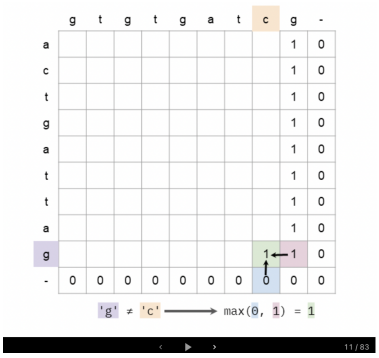


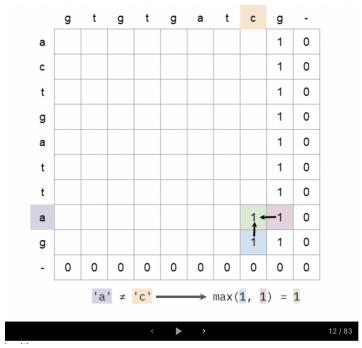

'g' = 'g' 1 + 0 = 1
7 / 83

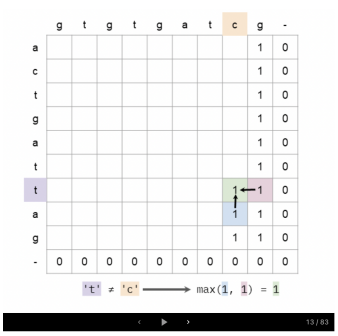

't' ≠ 'g' max(1, 0) = 1
8 / 83

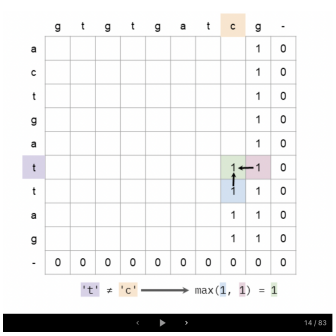

'c' ≠ 'g' max(1, 0) = 1
9 / 83

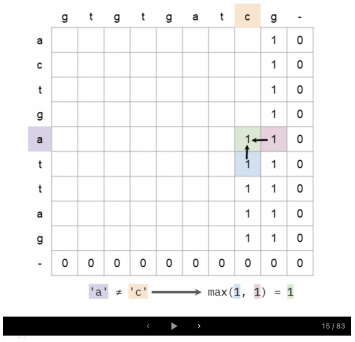

'a' ≠ 'g' max(1, 0) = 1
10 / 83

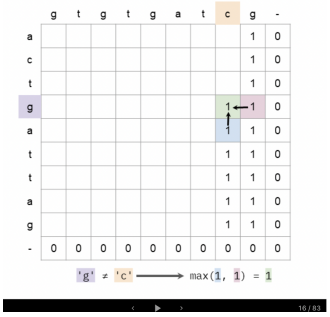

'g' ≠ 'c' max(0, 1) = 1
11 / 83

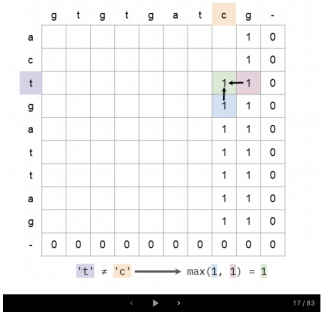

'a' ≠ 'c' max(1, 1) = 1
12 / 83

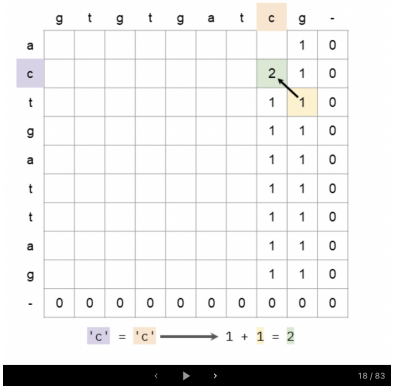

't' ≠ 'c' max(1, 1) = 1
13 / 83

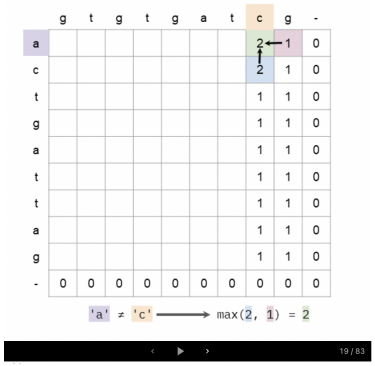

't' ≠ 'c' max(1, 1) = 1
14 / 83

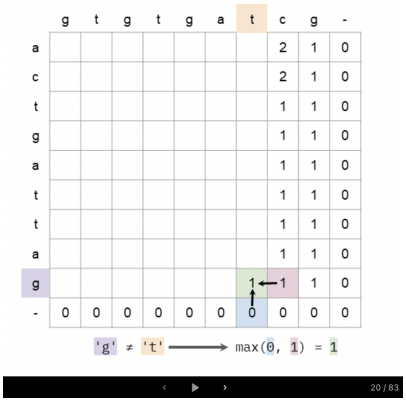

'a' ≠ 'c' max(1, 1) = 1
15 / 83

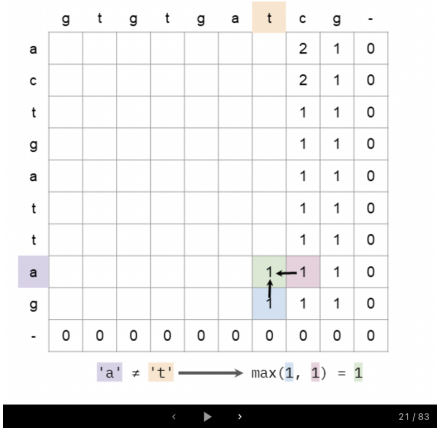

'g' ≠ 'c' max(1, 1) = 1
16 / 83

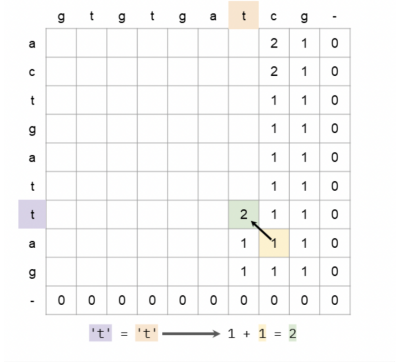

't' ≠ 'c' max(1, 1) = 1
17 / 83

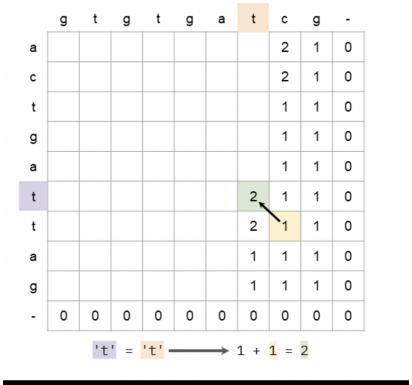

'c' = 'c' 1 + 1 = 2
18 / 83

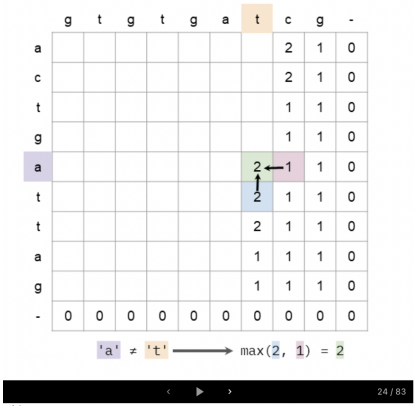

'a' ≠ 'c' ⟶ max(2, 1) = 2
19 / 83

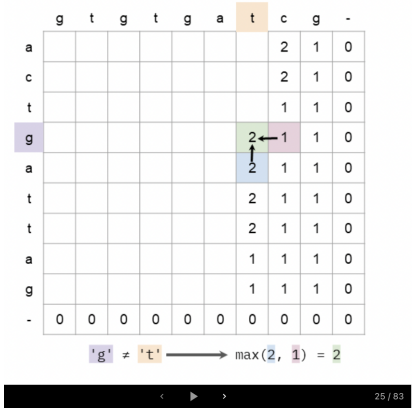

'g' ≠ 't' ⟶ max(0, 1) = 1
20 / 83

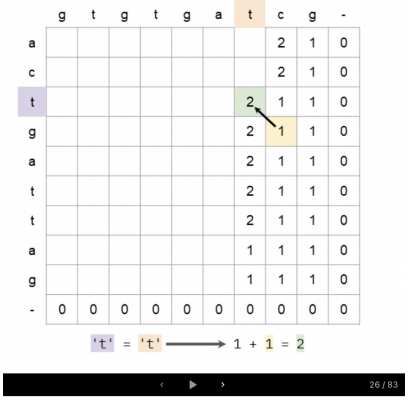

'a' ≠ 't' ⟶ max(1, 1) = 1
21 / 83

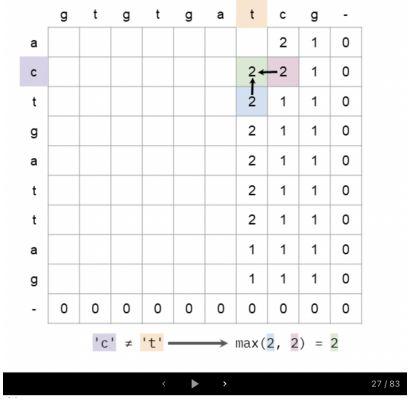

't' = 't' ⟶ 1 + 1 = 2
22 / 83

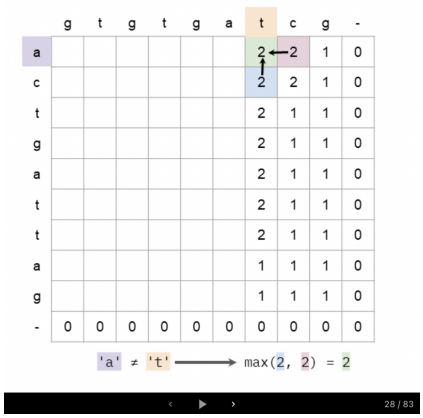

't' = 't' ⟶ 1 + 1 = 2
23 / 83

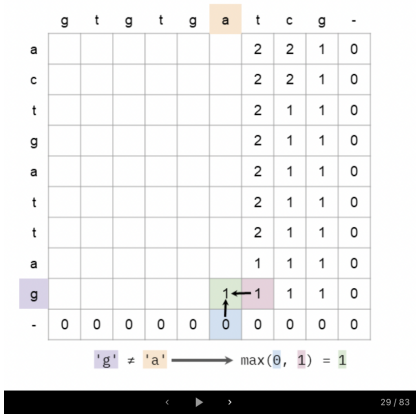

'a' ≠ 't' ⟶ max(2, 1) = 2
24 / 83

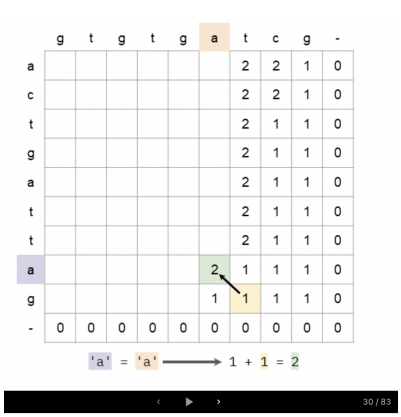

'g' ≠ 't' ⟶ max(2, 1) = 2
25 / 83

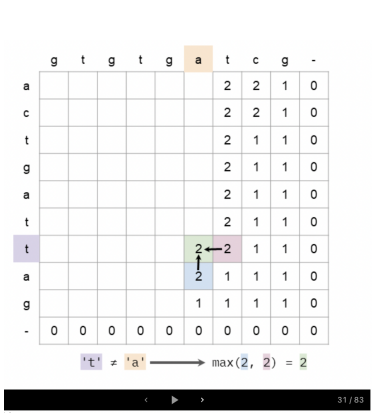

't' = 't' ⟶ 1 + 1 = 2
26 / 83

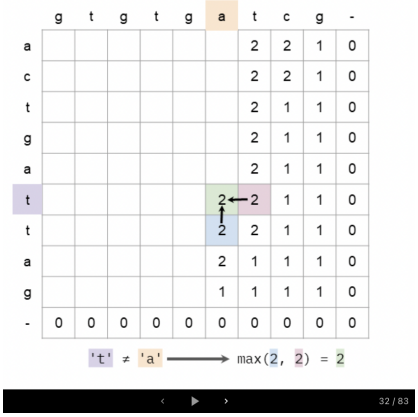

'c' ≠ 't' ⟶ max(2, 2) = 2
27 / 83

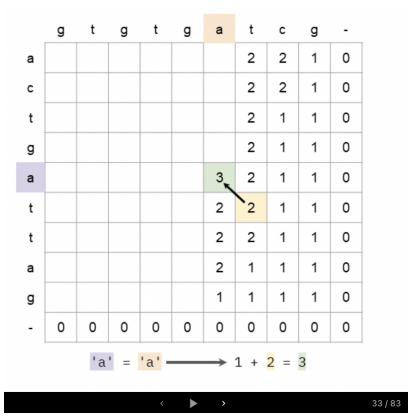

'a' ≠ 't' ⟶ max(2, 2) = 2
28 / 83

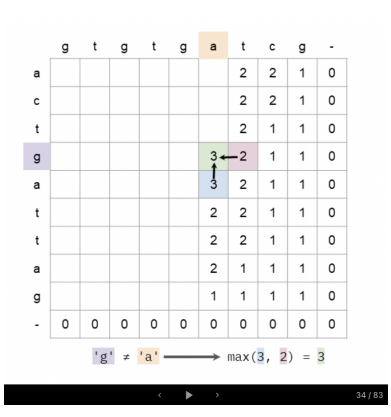

'g' ≠ 'a' ⟶ max(0, 1) = 1
29 / 83

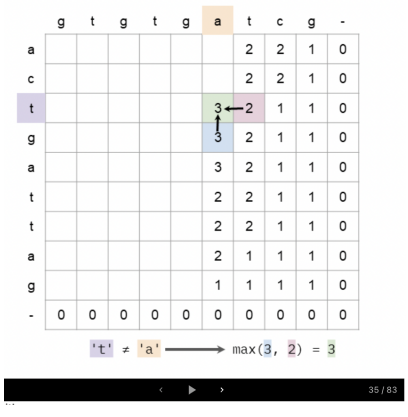

'a' = 'a' ⟶ 1 + 1 = 2
30 / 83

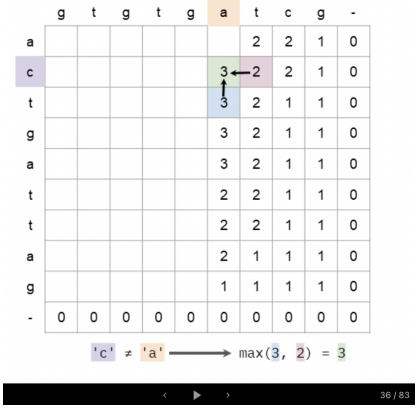

't' ≠ 'a' ⟶ max(2, 2) = 2
31 / 83

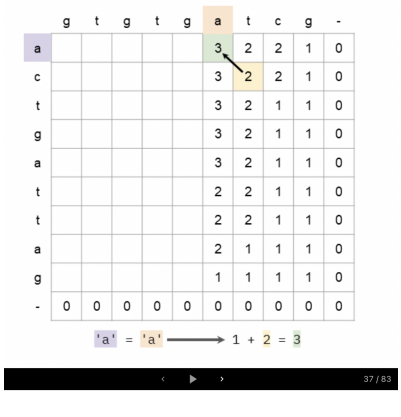

't' ≠ 'a' ⟶ max(2, 2) = 2
32 / 83

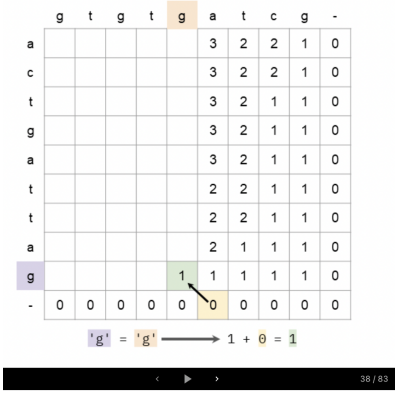

'a' = 'a' ⟶ 1 + 2 = 3
33 / 83

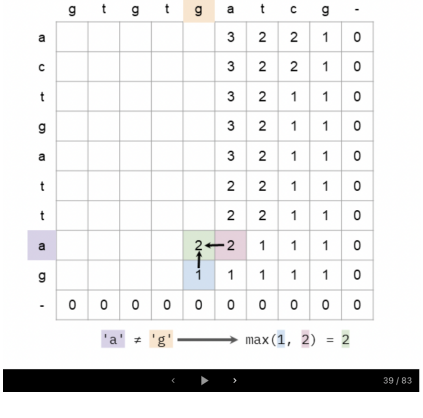

'g' ≠ 'a' ⟶ max(3, 2) = 3
34 / 83

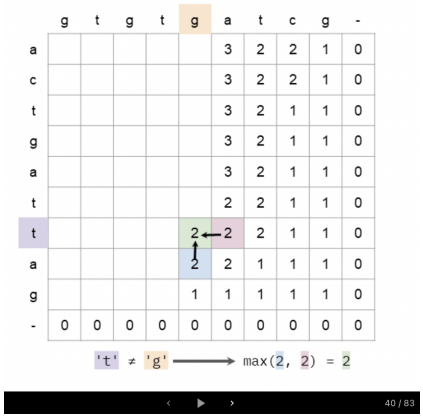

't' ≠ 'a' ⟶ max(3, 2) = 3
35 / 83

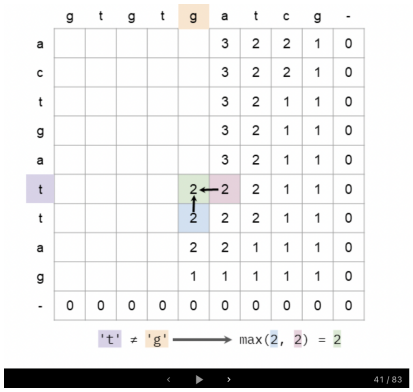

'c' ≠ 'a' ⟶ max(3, 2) = 3
36 / 83

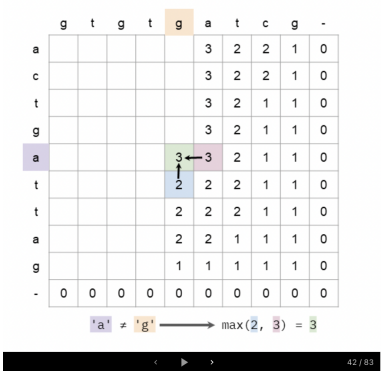

'a' = 'a' ⟶ 1 + 2 = 3
37 / 83

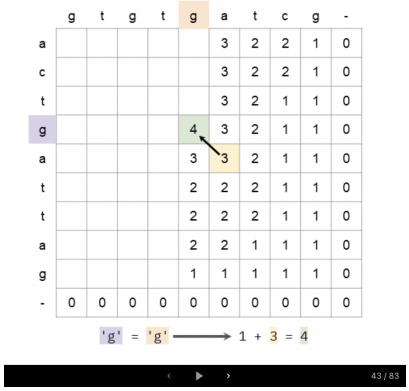

'g' = 'g' ⟶ 1 + 0 = 1
38 / 83

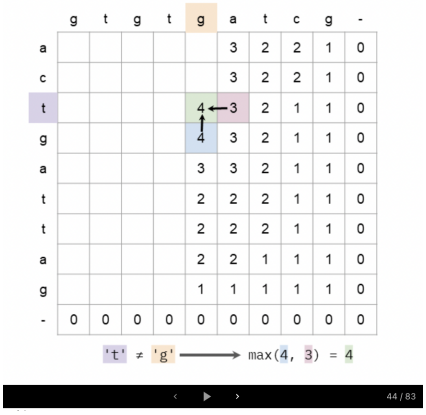

'a' ≠ 'g' ⟶ max(1, 2) = 2
39 / 83

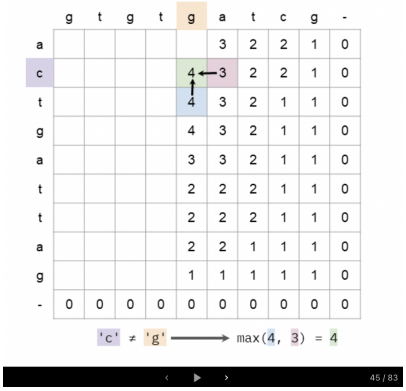

't' ≠ 'g' ⟶ max(2, 2) = 2
40 / 83

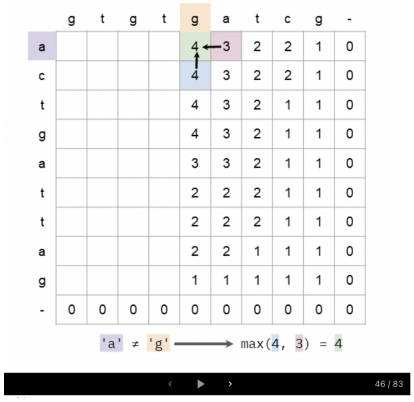

't' ≠ 'g' ⟶ max(2, 2) = 2
41 / 83

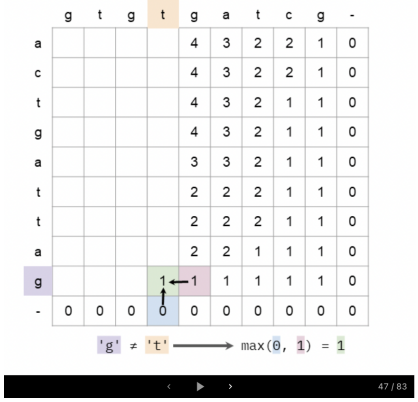

'a' ≠ 'g' ⟶ max(2, 3) = 3
42 / 83

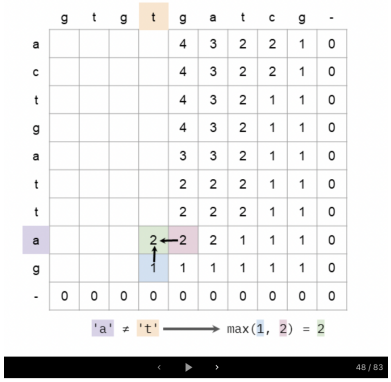

'g' = 'g' ⟶ 1 + 3 = 4
43 / 83

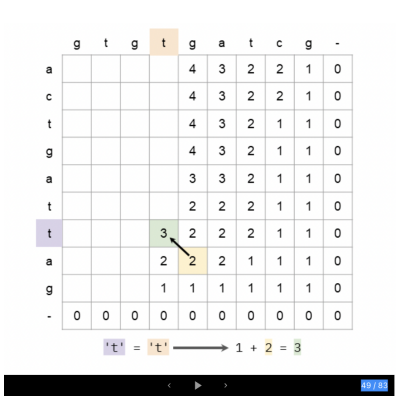

't' ≠ 'g' ⟶ max(4, 3) = 4
44 / 83

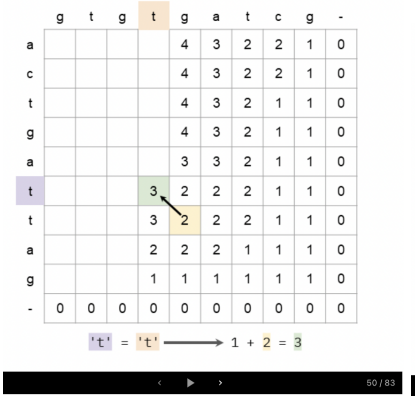

'c' ≠ 'g' ⟶ max(4, 3) = 4
45 / 83

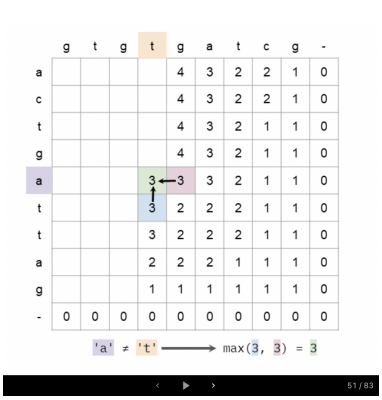

'a' ≠ 'g' ⟶ max(4, 3) = 4
46 / 83

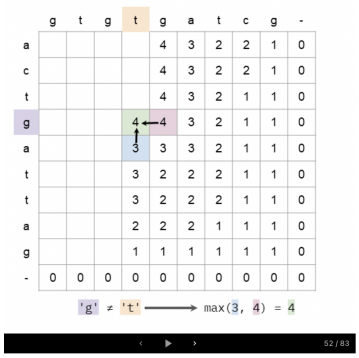

'g' ≠ 't' ⟶ max(0, 1) = 1
47 / 83

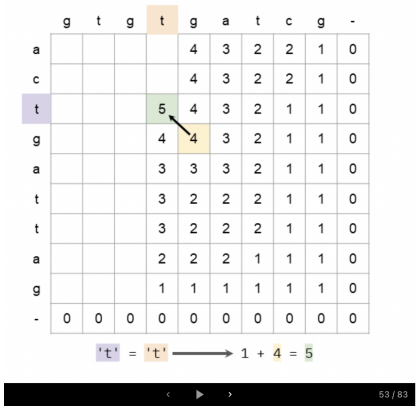

'a' ≠ 't' ⟶ max(1, 2) = 2
48 / 83

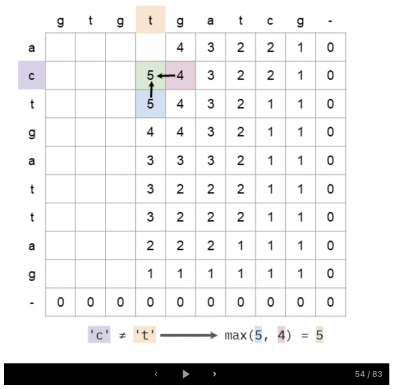

't' = 't' ⟶ 1 + 2 = 3
49 / 83

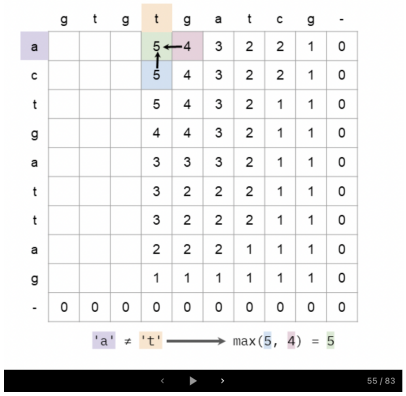

't' = 't' ⟶ 1 + 2 = 3
50 / 83

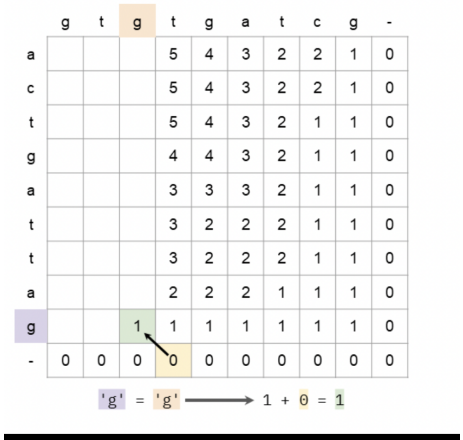

'a' ≠ 't' ⟶ max(3, 3) = 3
51 / 83

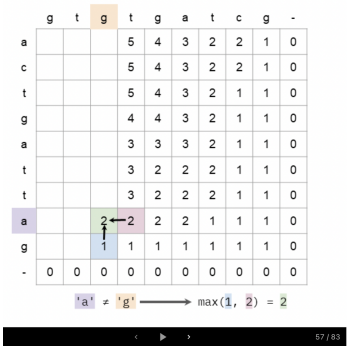

'g' ≠ 't' ⟶ max(3, 4) = 4
52 / 83

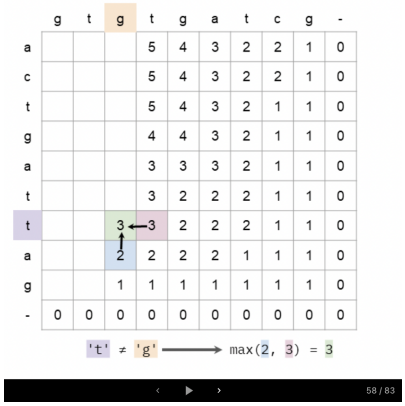

't' = 't' ⟶ 1 + 4 = 5
53 / 83

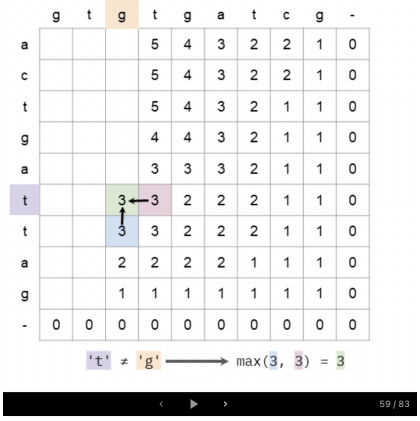

'c' ≠ 't' ⟶ max(5, 4) = 5
54 / 83

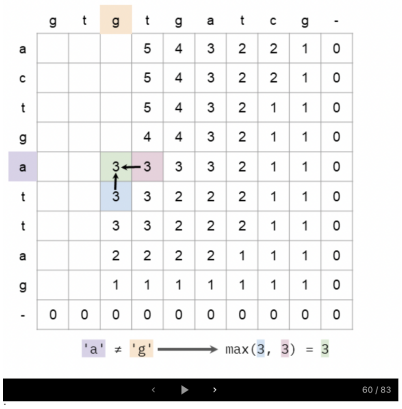

'a' ≠ 't' ⟶ max(5, 4) = 5
55 / 83

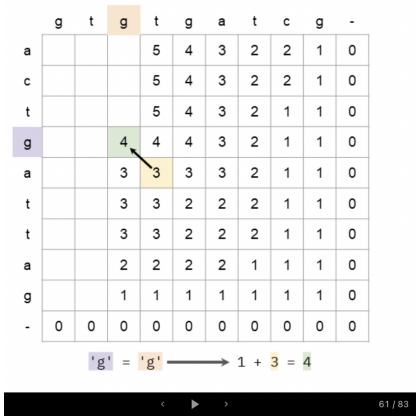

'g' = 'g' ⟶ 1 + 0 = 1
56 / 83

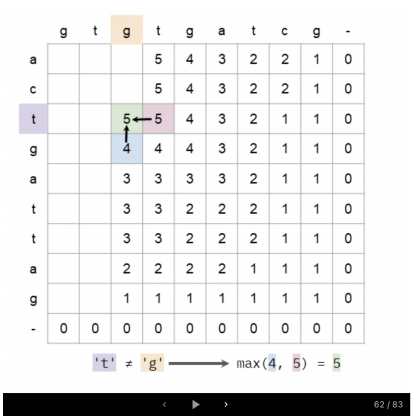

'a' ≠ 'g' ⟶ max(1, 2) = 2
57 / 83

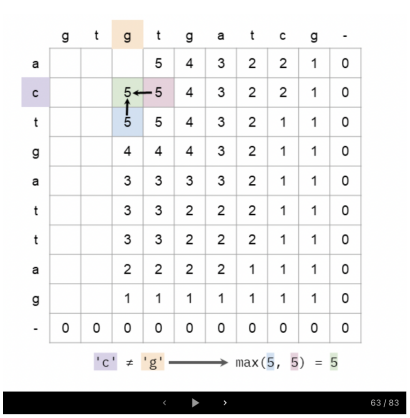

't' ≠ 'g' ⟶ max(2, 3) = 3
58 / 83

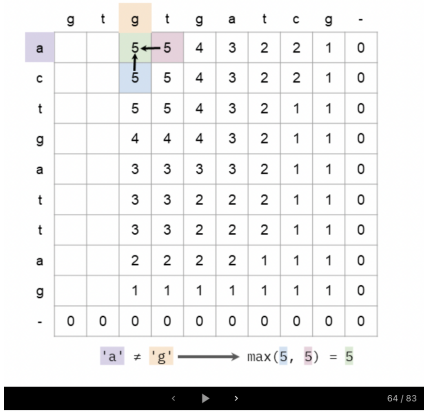

't' ≠ 'g' ⟶ max(3, 3) = 3
59 / 83

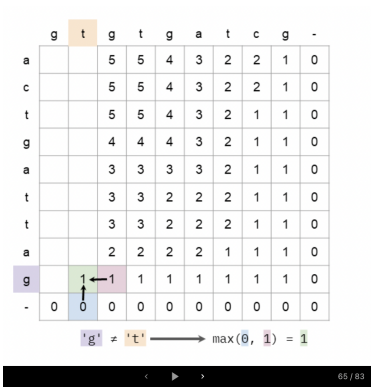

'a' ≠ 'g' ⟶ max(3, 3) = 3
60 / 83

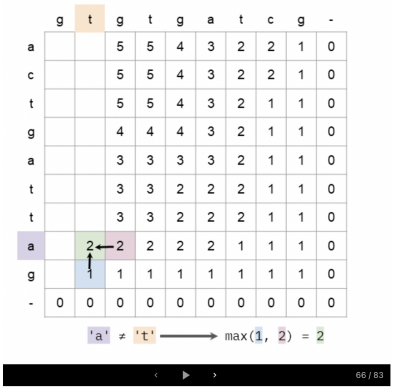

g t g t g a t c g -
a 5 4 3 2 2 1 0
c 5 4 3 2 2 1 0
t 5 4 3 2 1 1 0
g 4 4 4 3 2 1 1 0
a 3 3 3 3 2 1 1 0
t 3 3 2 2 2 1 1 0
t 3 3 2 2 2 1 1 0
a 2 2 2 2 1 1 1 0
g 1 1 1 1 1 1 1 0
- 0 0 0 0 0 0 0 0 0
'g' = 'g' ⟶ 1 + 3 = 4
61 / 83

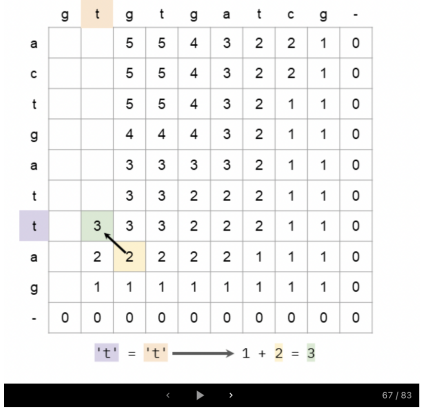

g t g t g a t c g -
a 5 4 3 2 2 1 0
c 5 4 3 2 2 1 0
t 5 5 4 3 2 1 1 0
g 4 4 4 3 2 1 1 0
a 3 3 3 3 2 1 1 0
t 3 3 2 2 2 1 1 0
t 3 3 2 2 2 1 1 0
a 2 2 2 2 1 1 1 0
g 1 1 1 1 1 1 1 0
- 0 0 0 0 0 0 0 0 0
't' ≠ 'g' ⟶ max(4, 5) = 5
62 / 83

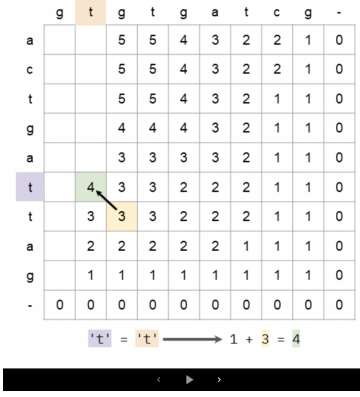

g t g t g a t c g -
a 5 4 3 2 2 1 0
c 5 5 4 3 2 2 1 0
t 5 5 4 3 2 1 1 0
g 4 4 4 3 2 1 1 0
a 3 3 3 3 2 1 1 0
t 3 3 2 2 2 1 1 0
t 3 3 2 2 2 1 1 0
a 2 2 2 2 1 1 1 0
g 1 1 1 1 1 1 1 0
- 0 0 0 0 0 0 0 0 0
'c' ≠ 'g' ⟶ max(5, 5) = 5
63 / 83

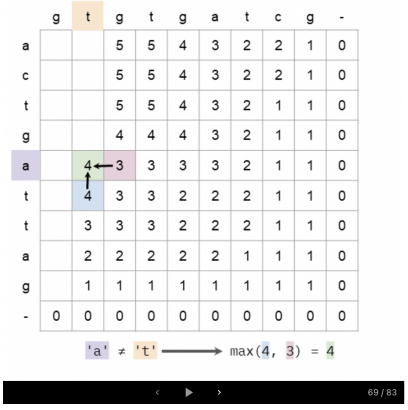

g t g t g a t c g -
a 5 5 4 3 2 2 1 0
c 5 5 4 3 2 2 1 0
t 5 5 4 3 2 1 1 0
g 4 4 4 3 2 1 1 0
a 3 3 3 3 2 1 1 0
t 3 3 2 2 2 1 1 0
t 3 3 2 2 2 1 1 0
a 2 2 2 2 1 1 1 0
g 1 1 1 1 1 1 1 0
- 0 0 0 0 0 0 0 0 0
'a' ≠ 'g' ⟶ max(5, 5) = 5
64 / 83

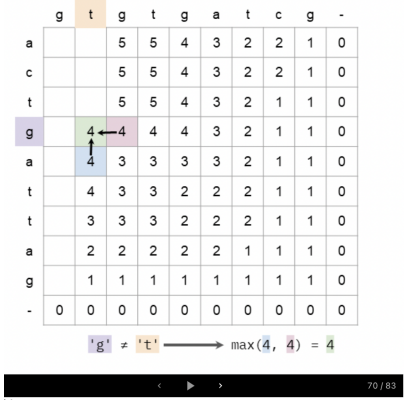

g t g t g a t c g -
a 5 5 4 3 2 2 1 0
c 5 5 4 3 2 2 1 0
t 5 5 4 3 2 1 1 0
g 4 4 4 3 2 1 1 0
a 3 3 3 3 2 1 1 0
t 3 3 2 2 2 1 1 0
t 3 3 2 2 2 1 1 0
a 2 2 2 2 1 1 1 0
g 1 1 1 1 1 1 1 0
- 0 0 0 0 0 0 0 0 0
'g' ≠ 't' ⟶ max(0, 1) = 1
65 / 83

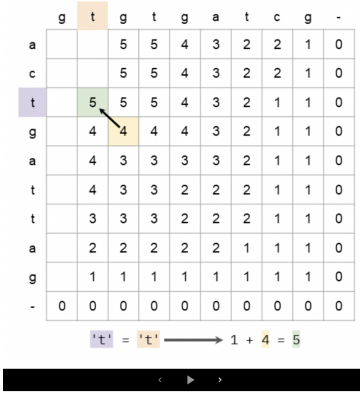

g t g t g a t c g -
a 5 5 4 3 2 2 1 0
c 5 5 4 3 2 2 1 0
t 5 5 4 3 2 1 1 0
g 4 4 4 3 2 1 1 0
a 3 3 3 3 2 1 1 0
t 3 3 2 2 2 1 1 0
t 3 3 2 2 2 1 1 0
a 2 2 2 2 1 1 1 0
g 1 1 1 1 1 1 1 0
- 0 0 0 0 0 0 0 0 0
'a' ≠ 't' ⟶ max(1, 2) = 2
66 / 83

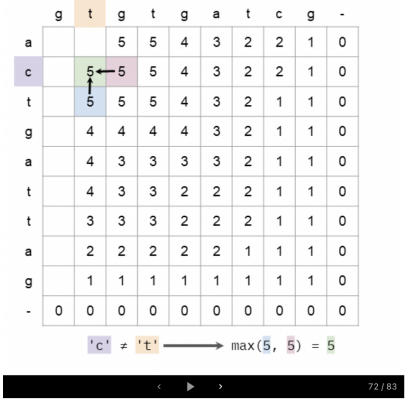

g t g t g a t c g -
a 5 5 4 3 2 2 1 0
c 5 5 4 3 2 2 1 0
t 5 5 4 3 2 1 1 0
g 4 4 4 3 2 1 1 0
a 3 3 3 3 2 1 1 0
t 3 3 2 2 2 1 1 0
t 3 3 2 2 2 1 1 0
a 2 2 2 2 1 1 1 0
g 1 1 1 1 1 1 1 0
- 0 0 0 0 0 0 0 0 0
't' = 't' ⟶ 1 + 2 = 3
67 / 83

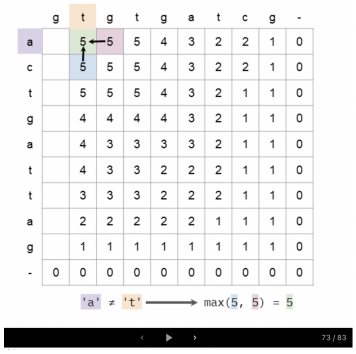

g t g t g a t c g -
a 5 5 4 3 2 2 1 0
c 5 5 4 3 2 2 1 0
t 5 5 4 3 2 1 1 0
g 4 4 4 3 2 1 1 0
a 3 3 3 3 2 1 1 0
t 4 3 3 2 2 2 1 1 0
t 3 3 3 2 2 2 1 1 0
a 2 2 2 2 2 1 1 1 0
g 1 1 1 1 1 1 1 1 0
- 0 0 0 0 0 0 0 0 0 0
't' = 't' ⟶ 1 + 3 = 4
68 / 83

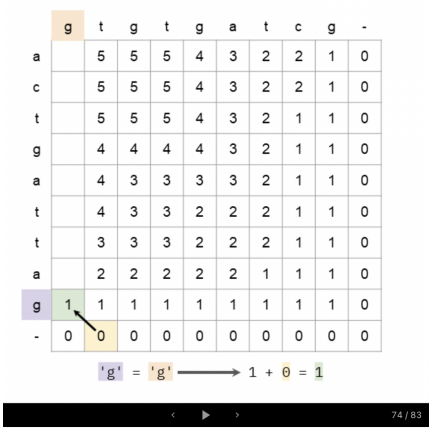

g t g t g a t c g -
a 5 5 4 3 2 2 1 0
c 5 5 4 3 2 2 1 0
t 5 5 4 3 2 1 1 0
g 4 4 4 3 2 1 1 0
a 4 3 3 3 3 2 1 1 0
t 4 3 3 2 2 2 1 1 0
t 3 3 3 2 2 2 1 1 0
a 2 2 2 2 2 1 1 1 0
g 1 1 1 1 1 1 1 1 0
- 0 0 0 0 0 0 0 0 0 0
'a' ≠ 't' ⟶ max(4, 3) = 4
69 / 83

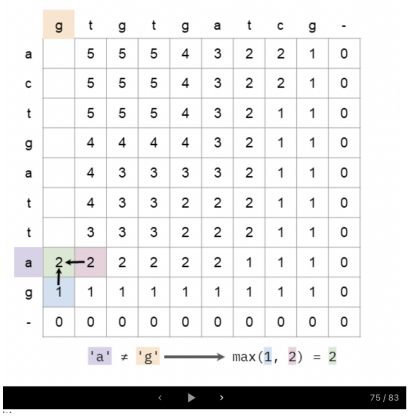

g t g t g a t c g -
a 5 5 4 3 2 2 1 0
c 5 5 4 3 2 2 1 0
t 5 5 4 3 2 1 1 0
g 4 4 4 3 2 1 1 0
a 4 3 3 3 3 2 1 1 0
t 4 3 3 2 2 2 1 1 0
t 3 3 3 2 2 2 1 1 0
a 2 2 2 2 2 1 1 1 0
g 1 1 1 1 1 1 1 1 0
- 0 0 0 0 0 0 0 0 0 0
'g' ≠ 't' ⟶ max(4, 4) = 4
70 / 83

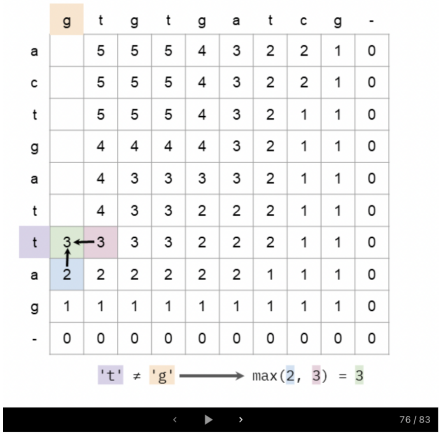

g t g t g a t c g -
a 5 5 4 3 2 2 1 0
c 5 5 4 3 2 2 1 0
t 5 5 5 4 3 2 1 1 0
g 4 4 4 4 3 2 1 1 0
a 4 3 3 3 3 2 1 1 0
t 4 3 3 2 2 2 1 1 0
t 3 3 3 2 2 2 1 1 0
a 2 2 2 2 2 1 1 1 0
g 1 1 1 1 1 1 1 1 0
- 0 0 0 0 0 0 0 0 0 0
't' = 't' ⟶ 1 + 4 = 5
71 / 83

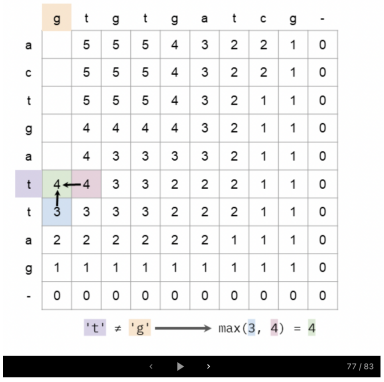

g t g t g a t c g -
a 5 5 4 3 2 2 1 0
c 5 5 4 3 2 2 1 0
t 5 5 5 4 3 2 1 1 0
g 4 4 4 4 3 2 1 1 0
a 4 3 3 3 3 2 1 1 0
t 4 3 3 2 2 2 1 1 0
t 3 3 3 2 2 2 1 1 0
a 2 2 2 2 2 1 1 1 0
g 1 1 1 1 1 1 1 1 0
- 0 0 0 0 0 0 0 0 0 0
'c' ≠ 't' ⟶ max(5, 5) = 5
72 / 83

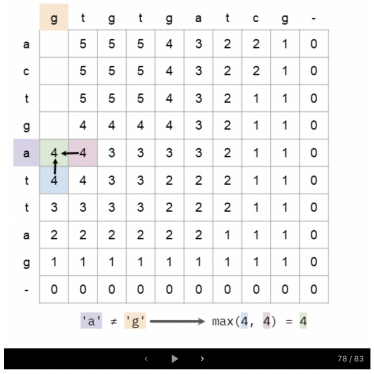

'a' ≠ 't' ⟶ max(5, 5) = 5
73 / 83

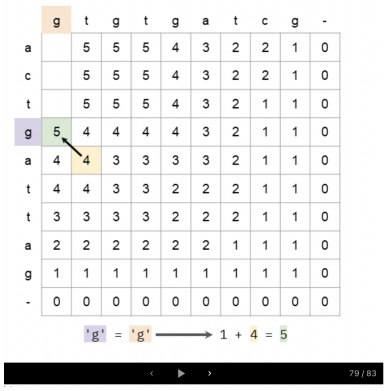

'g' = 'g' ⟶ 1 + 0 = 1
74 / 83

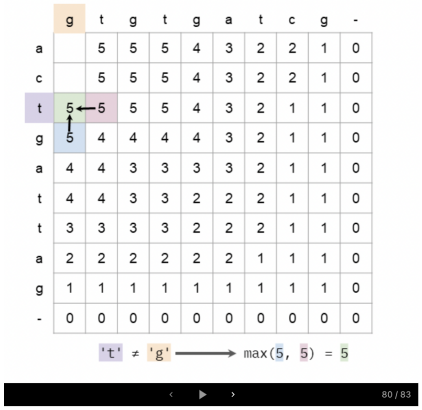

'a' ≠ 'g' ⟶ max(1, 2) = 2
75 / 83

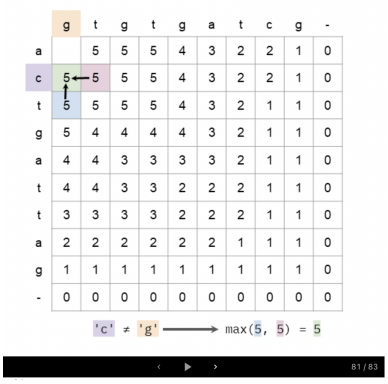

't' ≠ 'g' ⟶ max(2, 3) = 3
76 / 83

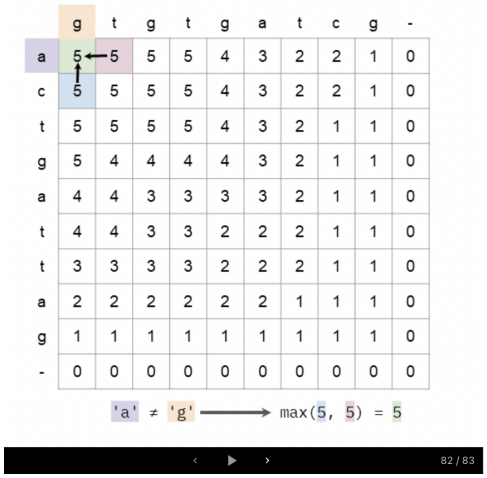

't' ≠ 'g' ⟶ max(3, 4) = 4
77 / 83

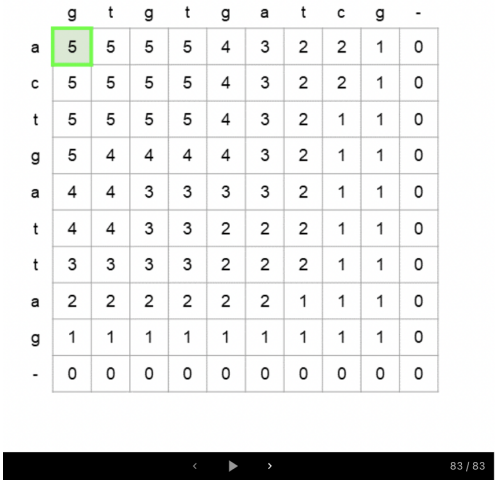

'a' ≠ 'g' ⟶ max(4, 4) = 4
78 / 83











Algorithm

Java | Python3

```java
class Solution {

  public int longestCommonSubsequence(String text1, String text2) {

    // Make a grid of 0's with text2.length() + 1 columns
    // and text1.length() + 1 rows.
    int[][] dpGrid = new int[text1.length() + 1][text2.length() + 1];

    // Iterate up each column, starting from the last one.
    for (int col = text2.length() - 1; col >= 0; col--) {
      for (int row = text1.length() - 1; row >= 0; row--) {
        // If the corresponding characters for this cell are the same...
        if (text1.charAt(row) == text2.charAt(col)) {
          dpGrid[row][col] = 1 + dpGrid[row + 1][col + 1];
        // Otherwise they must be different...
        } else {
          dpGrid[row][col] = Math.max(dpGrid[row + 1][col], dpGrid[row][col + 1]);
        }
      }
    }

    // The original problem's answer is in dp_grid[0][0]. Return it.
    return dpGrid[0][0];
  }
}
```

Complexity Analysis

- Time complexity : $O(M \cdot N)$.

  We're solving $M \cdot N$ subproblems. Solving each subproblem is an $O(1)$ operation.

- Space complexity : $O(M \cdot N)$.

  We'e allocating a 2D array of size $M \cdot N$ to save the answers to subproblems.

Approach 4: Dynamic Programming with Space Optimization

Intuition

You might have noticed in the Approach 3 animation that we only ever looked at the current column and the previous column. After that, previously computed columns are no longer needed (if you didn't notice, go back and look at the animation).

> We can save a lot of space by instead of keeping track of an entire 2D array, only keeping track of the last two columns.

This reduces the space complexity to be proportional to the length of the word going down. We should make sure this is the shortest of the two words.

Algorithm

Java | Python3 — Copy

```java
class Solution {

  public int longestCommonSubsequence(String text1, String text2) {

    // If text1 doesn't reference the shortest string, swap them.
    if (text2.length() < text1.length()) {
      String temp = text1;
      text1 = text2;
      text2 = temp;
    }

    // The previous column starts with all 0's and like before is 1
    // more than the length of the first word.
    int[] previous = new int[text1.length() + 1];

    // Iterate through each column, starting from the last one.
    for (int col = text2.length() - 1; col >= 0; col--) {
      // Create a new array to represent the current column.
      int[] current = new int[text1.length() + 1];
      for (int row = text1.length() - 1; row >= 0; row--) {
        if (text1.charAt(row) == text2.charAt(col)) {
          current[row] = 1 + previous[row + 1];
        } else {
          current[row] = Math.max(previous[row], current[row + 1]);
        }
      }
      // The current column becomes the previous one.
```

We can still do better! Thanks @heinzerm for bringing this up in the comments :)

One slight inefficiency in the above code is that a new `current` array is created on each loop cycle. While this doesn't affect the space *complexity*—as we assume garbage collection happens immediately for the purposes of space complexity analysis—it does improve the *actual* time and space usage by a *constant* amount.

A couple of people have suggested that we could create `current` in the same place we create `previous`. Then each time, the `current` array will be reused. This, they argue, should work because we never modify the `0` at the end (the padding), and then, other than that `0`, we're only ever *reading* from indexes that we've *already written to on that outer-loop iteration*. This logic is entirely correct.

However, it will break for a different reason. The line `previous = current` makes both `previous` and `current` reference ***the same*** *list*. This happens at the end of the first loop cycle. So after that point, the algorithm is no longer functioning as intended.

There is another solution, though: notice that when we do `previous = current`, we are removing the reference to the `previous` array. At this point, it would normally be garbage collected. Instead, though, we could use that array as our `current` array for the next iteration! This way, we're not making both variables reference the same array, and we're reusing a no longer array instead of creating a new one. Correctness is guaranteed, as explained above, we're only ever reading the `0` at the end or values we've already written to in that outer-loop iteration.

Here is the slightly modified code for your reference.

```java
class Solution {

  public int longestCommonSubsequence(String text1, String text2) {

    // If text1 doesn't reference the shortest string, swap them.
    if (text2.length() < text1.length()) {
      String temp = text1;
      text1 = text2;
      text2 = temp;
    }

    // The previous and current column starts with all 0's and like
    // before is 1 more than the length of the first word.
    int[] previous = new int[text1.length() + 1];
    int[] current = new int[text1.length() + 1];

    // Iterate through each column, starting from the last one.
```

```
    for (int col = text2.length() - 1; col >= 0; col--) {
      for (int row = text1.length() - 1; row >= 0; row--) {
        if (text1.charAt(row) == text2.charAt(col)) {
          current[row] = 1 + previous[row + 1];
        } else {
          current[row] = Math.max(previous[row], current[row + 1]);
        }
      }
      // The current column becomes the previous one, and vice versa.
      int[] temp = previous;
      previous = current;
      current = temp;
    }

    // The original problem's answer is in previous[0]. Return it.
    return previous[0];
  }
}
```

Complexity Analysis

Let M be the length of the first word, and N be the length of the second word.

- Time complexity : $O(M \cdot N)$.

  Like before, we're solving $M \cdot N$ subproblems, and each is an $O(1)$ operation to solve.

- Space complexity : $O(\min(M, N))$.

  We've reduced the auxilary space required so that we only use two 1D arrays at a time; each the length of the shortest input word. Seeing as the 2 is a constant, we drop it, leaving us with the minimum length out of the two words.