# Dynamic Programming

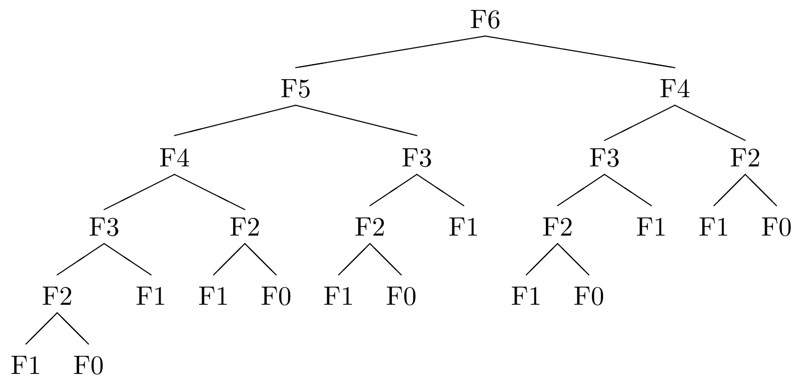
Due: April 16, 11:55 PM

## Overview

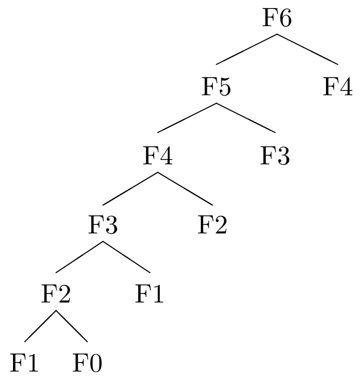
In this assignment you will be solving several problems requiring the use of dynamic programming. Dynamic programming is an algorithmic technique which is usually based off of a recurrent formula, and some amount of starting states. Answers to sub-problems are constructed from smaller sub-problems (initially the starting states) until you have built up to your final answer. For example consider the famous Fibonacci sequence:

$$F(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ F(n-2) + F(n-1) & \text{if } n \geq 2 \end{cases}$$

Here we have two starting states, and a recurrence relation. If we were to naively solve this using recursion we could end up with a solution with time complexity $O(2^n)$. Using this implementation calculating $F(100)$ on a computer able to perform $1,000,000,000,000$ operations per second would take approximately 40 billion years. Fortunately, using dynamic programming we can do *much* better. If we consider the recursion tree of the naive implementation, $F(6)$ will look as follows.



Here we are calculating $F(4)$ twice, $F(3)$ three times, and $F(2)$ five times. This tree will grow exponentially. If we try using a dynamic programming technique where we save the answers to our subproblems each time we calculate them, we can then perform a simple lookup to determine the answer of $F(n)$, after we have calculated it once. This will result in the following tree.



This tree will grow at a linear rate, rather than exponential, and makes Fibonacci much easier to solve for larger values. The concept of saving your subproblems as you calculate them is one of the main concepts of dynamic programming, and as you can see, it has a large impact on running time. Improving an algorithm that was $O(2^n)$ to $O(n)$.

## Longest Common Subsequence

In this problem you will be given two strings, your task is to find the longest common subsequence within these two strings. Consider the strings: `"apple cider"`, and `"please"`. The longest common subsequence for this example would be `"plee"`. Now lets try to develop a dynamic programming solution for this problem.

We have two strings, $A$ and $B$, of length $m$ and $n$ respectively. We will define the function $LCS(i, j)$ as the solution to the subproblem of the longest common subsequence of string $A$ from $0 \to i,\ \ 0 \leq i \leq m$, and of string $B$ from $0 \to j,\ \ 0 \leq j \leq n$.

Our goal is to solve $LCS(m, n)$ which can be done if we know the solution to $LCS(m-1, n-1)$, $LCS(m-1, n)$, and $LCS(m, n-1)$. If string $A$ at the $m_{th}$ character is the same character as string $B$ at the $n_{th}$ character, then we know that we can increase the length of the solution to $LCS(m-1, n-1)$ by one, because the next characters in each string match. Otherwise, the characters don't match and we will simply take the maximum of $LCS(m-1, n)$ and $LCS(m, n-1)$ to be the solution to $LCS(m, n)$. We now have a well defined set of subproblems we can use to build up to $LCS(m, n)$.

Determining our starting states is the next step. Lets look at $LCS(0, j)$ the answer will always be 0, because the empty string has no characters in common with any other string. Similarly $LCS(i, 0)$ will be 0 for all values of $i$. Now we have both our starting states, and a method for building up from our initial subproblems.

When two characters match, at $(i, j)$, it will take the value of the diagonal to the upper-left, $(i-1, j-1)$, plus 1. If they do not match, we take the max of the value above it, and the value to the left as $(i, j)$'s value. Visually the algorithm will look as follows, for the strings: "sharpie" and "grape".

Initialization

| | | s | h | a | r | p | i | e |
|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| g | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| r | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| a | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| p | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| e | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

R is the first match

| | | s | h | a | r | p | i | e |
|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| g | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| r | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| a | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| p | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| e | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Fill out the A row

| | | s | h | a | r | p | i | e |
|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| g | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| r | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| a | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| p | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| e | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Fill out the P row

| | | s | h | a | r | p | i | e |
|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| g | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| r | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| a | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| p | 0 | 0 | 0 | 1 | 1 | 2 | 2 | 2 |
| e | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Fill out the E row

| | | s | h | a | r | p | i | e |
|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| g | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| r | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| a | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| p | 0 | 0 | 0 | 1 | 1 | 2 | 2 | 2 |
| e | 0 | 0 | 0 | 1 | 1 | 2 | 2 | 3 |

Final, LCS has a length of 3

| | | s | h | a | r | p | i | e |
|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| g | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| r | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| a | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| p | 0 | 0 | 0 | 1 | 1 | 2 | 2 | 2 |
| e | 0 | 0 | 0 | 1 | 1 | 2 | 2 | 3 |

### Pseudo-code

```
String str1;
String str2;
int[][] array = ...  //set up array of appropriate size

for (i = 0 to array.length - 1)
    //set up base cases
for (i = 0 to array[0].length - 1)
    //set up base cases

for (j = 1 to array[0].length - 1)
    for (i = 1 to array.length - 1)
        if (str1.charAt(i - 1) == str2.charAt(j - 1))
            //put appropriate value into array[i][j]
        else
            //put appropriate value into array[i][j]

return ...; // return appropriate value here
```

### Input

The input will consist of two Strings $A$ and $B$ of lengths $m$ and $n$ respectively. Where $1 \leq n, m \leq 5000$ the Strings will consist of only valid ASCII characters, note that **A** and **a** are not the same character, uppercase letters should not be treated as equal to their lowercase counterparts.

### Output

You should output the length of the Longest Common Subsequence of Strings $A$ and $B$, which should be the value located at:

```
array[array.length - 1][array[0].length - 1]
```

### Example

| Input | Output |
|---|---|
| `sharpie`<br>`grape` | 3 |
| `six`<br>`Seven` | 0 |
| `The brown dog ate an apple yesterday.`<br>`Once upon a time, the end.` | 14 |

## Edit Distance

In this problem you will be given two strings, $A$ and $B$, you will be finding the edit distance of these two strings. We will define the edit distance of two strings to be the minimum number of operations that must be performed on either string to make them the same.

For this problem there will be 3 operations we can perform:
Insert($i$, $c$): Insert the character $c$ into a string at position $i$
Remove($i$): Remove the character at position $i$ from a string
Change($i$, $c$): Change the character at position $i$ in a string to the character $c$

This problem will be very similar to the Longest Common Subsequence problem. We will define the subproblem edit($i$, $j$) to be the minimum number of operations required to match the first $i$ characters from $A$ to the first $j$ characters from $B$. This will be our subproblem, that we use to build up to our final answer. However, in this problem we are minimizing the entries in our array rather than maximizing them as we did in the last problem.

If we are considering the element in the array representing edit($i$, $j$) then there are once again 3 subproblems we need to consider: edit($i-1$, $j$), edit($i$, $j-1$) and edit($i-1$, $j-1$). Now we need to choose the minimum of all possible ways to reach edit($i$, $j$). From edit($i-1$, $j$) and edit($i$, $j-1$) it is easy to see that exactly 1 operation is required to reach edit($i$, $j$). Additionally if we consider the subproblem edit($i-1$, $j-1$) then if the characters at $i$ and $j$ are the same then no operations are required to reach edit($i$, $j$) because the next characters are the same. However if they are different then we must change one of these characters, which requires 1 operation. Considering all of these ways to reach the subproblem edit($i$, $j$) we end up with this relation:

$\text{edit}(i, j) = \min(\text{edit}(i-1, j) + 1, \text{edit}(i, j-1) + 1, \text{edit}(i-1, j-1) + \text{different}(i, j))$
where different($i$, $j$) is equal to 0 if characters $i$ and $j$ are the same, 1 otherwise.

All that is left to this problem is figuring out the base cases. How many operations are required to change the empty string into another string? Use this to fill out edit($i$, 0) for all $i$'s and edit(0,$j$) for all $j$'s.

### Input

The input will consist of two Strings $A$ and $B$ of lengths $m$ and $n$ respectively. Where $1 \leq n, m \leq 5000$ the Strings will consist of only valid ASCII characters, note that **A** and **a** are not the same character, uppercase letters should not be treated as equal to their lowercase counterparts.

### Output

You shuld output the Edit Distance of Strings $A$ and $B$, which should be the value located at:

```
array[array.length - 1][array[0].length - 1]
```

### Example

| Input | Output |
|---|---|
| `ababb`<br>`bbab` | 2 |
| `sally sells seashells`<br>`by the seashore` | 13 |

## Deliverables

Please submit the following file. Make sure you run the provided JUnits to test for correctness..

- DynamicProgramming.java