CS 341: ALGORITHMS (S18) — LECTURE 9
DYNAMIC PROGRAMMING II: COMPARING STRINGS

ERIC BLAIS

In this lecture, we continue our exploration of the dynamic programming technique by examining problems related to comparisons of strings.

1. Longest common subsequence

A *common subsequence* between two strings $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$ is a string $z_1, \ldots, z_k$ for which there are indices $1 \leq i_1 < i_2 < \cdots < i_k \leq m$ and $1 \leq j_1 < j_2 < \cdots < j_k \leq n$ where for each $\ell \leq k$, $z_\ell = x_{i_\ell} = y_{j_\ell}$. For example, in the pair of strings

$$x = \texttt{POLYNOMIAL}$$
$$y = \texttt{EXPONENTIAL},$$

the string $z = \texttt{PONIAL}$ is a subsequence of x and y. (As are the strings `POL`, `PNA`, etc.)

Definition 9.1 (Longest common subsequence)**.** An instance of the *longest common subsequence (LCS)* problem is a pair of strings $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$. The valid solution to an instance is the length of the longest common subsequence of x and y.

The natural way to break down the LCS problem into smaller subproblems is to consider the longest common subsequence of prefixes of x and y. For $0 \leq i \leq m$ and $0 \leq j \leq n$, define

$$M(i,j) = \text{length of LCS of } x_1, \ldots, x_i \text{ and } y_1, \ldots, y_j.$$

When i or j is 0 (which corresponds to x or y being an empty string), then

$$M(0,j) = 0 \qquad \text{and} \qquad M(i,0) = 0.$$

Given that we have computed $M(i-1,j)$, $M(i,j-1)$, and $M(i-1,j-1)$, can we now compute $M(i,j)$? Indeed we can!

$$M(i,j) = \max \begin{cases} M(i-1,j-1)+1 & \text{if } x_i = y_j \\ M(i-1,j) \\ M(i,j-1) \end{cases}$$

This gives the following algorithm.

Algorithm 1: LCS$(x_1, \ldots, x_m, y_1, \ldots, y_n)$

```
for i = 1, ..., m do M[i, 0] = 0;
for j = 1, ..., n do M[0, j] = 0;
for i = 1, ..., m do
    for j = 1, ..., n do
        M[i, j] = max{M[i − 1, j], M[i, j − 1]};
        if x_i = y_j then M[i, j] = max{M[i, j], M[i − 1, j − 1] + 1};
return M[m, n];
```

We can picture the algorithm as filling out the table of values $M[i, j]$, row by row. With the instance $x =$ `ALGORITHM`, $y =$ `ANALYSIS`, the table looks as follows.

	∅	A	N	A	L	Y	S	I	S
∅	0	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1	1
L	0	1	1	1	2	2	2	2	2
G	0	1	1	1	2	2	2	2	2
O	0	1	1	1	2	2	2	2	2
R	0	1	1	1	2	2	2	2	2
I	0	1	1	1	2	2	2	3	3
T	0	1	1	.	.	.	.	.	.
H	0	.	.	.	.	.	.	.	.
M	0	.	.	.	.	.	.	.	.

The time complexity of the algorithm is $\Theta(mn)$.

2. Edit distance

The length of the longest common subsequence can be interpreted as a measure of how similar two strings are. A more sophisticated measure of the similarity between different strings is known as *edit distance*.

Definition 9.2 (Edit distance)**.** The *edit distance* between two strings $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$ is the minimum number of edit operations required to transform x into y, where the 3 possible edit operations are:

Adding: a letter to x,
Deleting: a letter from x, and
Replacing: a letter in x with another one.

For example, the edit distance between the strings `POLYNOMIAL` and `EXPONENTIAL` is 6, as this is the minimum number of edit operations required to go from one string to the other:

```
--POLYNOMIAL
EXPONEN-TIAL
```

In this example, the full list of operations that transformed `POLYNOMIAL` into `EXPONENTIAL` is as follows.

POLYNOMIAL	
EPOLYNOMIAL	(Add E)
EXPOLYNOMIAL	(Add X)
EXPONYNOMIAL	(Replace L with N)
EXPONENOMIAL	(Replace Y with E)
EXPONEN-MIAL	(Delete O)
EXPONENTIAL	(Replace M with T)

A basic computational problem is to find the edit distance between two strings.

Definition 9.3 (Edit distance problem)**.** An instance of the *edit distance problem* is two strings $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$; the valid solution to an instance is the edit distance between x and y.

We can again use the dynamic programming method to solve this problem by considering the subproblems obtained by computing the edit distance between prefixes of x and y. For $0 \leq i \leq m$ and $0 \leq j \leq n$, define

$$M(i, j) = \text{edit distance between } x_1, \ldots, x_i \text{ and } y_1, \ldots, y_j.$$

When $i = 0$ or $j = 0$, the edit distance is easy to compute: it is exactly the length of the other string. So

$$M(i, 0) = i \qquad \text{and} \qquad M(0, j) = j.$$

What about for the other entries? If $x_i = y_j$, then we can match those characters together and we get $M(i, j) = M(i - 1, j - 1)$. If not, we have three choices:

(1) Replace x_i with y_j. In this case, we get $M(i, j) = M(i - 1, j - 1) + 1$.
(2) Delete x_i. With this choice, $M(i, j) = M(i - 1, j) + 1$.
(3) Add the character y_j to the string x right before x_i. This choice gives us $M(i, j) = M(i, j - 1) + 1$.

To compute $M(i, j)$, we want to choose the option among the ones above that has minimum value. So this means that we get

$$M(i, j) = \max \begin{cases} M(i - 1, j - 1) & \text{if } x_i = y_j \\ M(i - 1, j - 1) + 1 & \text{if } x_i \neq y_j \\ M(i - 1, j) + 1 & \\ M(i, j - 1) + 1. & \end{cases}$$

As long as we compute the values of M in an order where $M(i - 1, j - 1)$, $M(i - 1, j)$, and $M(i, j - 1)$ have all been computed before we compute $M(i, j)$, computing this value takes $\Theta(1)$ time. We can again proceed row by row, obtaining an algorithm that looks very similar to the one we used to compute the length of the longest common subsequence.

Algorithm 2: EDITDISTANCE$(x_1, \ldots, x_m, y_1, \ldots, y_n)$

```
for i = 1, ..., m do M[i, 0] = i;
for j = 1, ..., n do M[0, j] = j;

for i = 1, ..., m do
    for j = 1, ..., n do
        if x_i = y_j then
            r = M[i − 1, j − 1];
        else
            r = M[i − 1, j − 1] + 1;
        M[i, j] = max{M[i − 1, j] + 1, M[i, j − 1] + 1, r};
return M[m, n];
```

The time complexity of this algorithm is again $\Theta(mn)$.

3. Finding the LCS

The LCS algorithm introduced in the first section determines the length of the longest common subsequence between the two strings given as input, but it does not identify the subsequence itself. What if we want to identify it? There are a number of different ways we can modify the algorithm to do so. Or, if we have already computed the matrix M of LCS lengths for all prefixes, we can identify the LCS itself by working backwards from $M(m, n)$.

Algorithm 3: PRINTLCS(x, y, M, i, j)

```
if i > 1 and M[i, j] = M[i − 1, j] then
    PrintLCS(x, y, M, i − 1, j);
else if j > 1 and M[i, j] = M[i, j − 1] then
    PrintLCS(x, y, M, i, j − 1);
else
    /* We must have matched x_i = y_j                    */
    PrintLCS(x, y, M, i − 1, j − 1);
    Print x_i;
```

Calling PRINTLCS(x, y, M, m, n) will print the longest common subsequence of x and y. Interestingly, by calling it with any other $i \leq m$ and $j \leq n$ we can also print the longest common subsequence of any prefixes of x and y just as efficiently.

A similar idea can be used to output a minimum set of edit operations that transform the string x to the string y after the EDITDISTANCE algorithm has been run. The details are left as an exercise.