Longest Common Subsequence

LCS is an interesting variation on the classical string matching problem: the task is that of finding the common portion of two strings (more precise definition in a couple of slides).

Applications[a]?

- Molecular biology. DNA sequences (genes) can be represented as sequences of four letters ACGT, corresponding to the four submolecules forming DNA. When biologists find a new sequence, they typically want to know what other sequences it is most similar to. One way of computing how similar two sequences are is to find the length of their longest common subsequence.

[a]Thanks to David Eppstein's web page:
`http://www.ics.uci.edu/ eppstein/161/960229.html+`

- File comparison. The Unix program `diff` is used to compare two different versions of the same file, to determine what changes have been made to the file. It works by finding a longest common subsequence of the lines of the two files; any line in the subsequence has not been changed, so what it displays is the remaining set of lines that have changed. In this instance of the problem we should think of each line of a file as being a single complicated character in a string.
- Screen redisplay. Many text editors like `emacs` display part of a file on the screen, updating the screen image as the file is changed. For slow dial-in terminals, these programs want to send the terminal as few characters as possible to cause it to update its display correctly. It is possible to view the computation of the minimum length sequence of characters needed to update the terminal as being a sort of common subsequence problem (the common subsequence tells you the parts of the display that are already correct and don't need to be changed).

Definitions

Given a sequence (i.e. a string of characters) $X = x_1 x_2 \ldots, x_m$, another sequence $Z = z_1 z_2 \ldots z_k$ is a *subsequence* of X if there exists a (strictly increasing) list of indices of X $i_1, i_2, \ldots, i_k$ such that for all $j \in \{1, 2, \ldots, k\}$, we have $x_{i_j} = z_j$.

Example. The sequence $Z = \mathsf{BCDB}$ is a subsequence of $X = \mathsf{ABCBDAB}$ with corresponding index list: $2, 3, 5, 7$.

The sequence $Z = \mathsf{1011}$ is a subsequence of $X = \mathsf{1011011}$ with corresponding index list: $1, 2, 3, 4$.

The empty sequence is a subsequence of all sequences.

Given two sequences X and Y, we say that a sequence Z is a *common subsequence* of X and Y if Z is a subsequence of both X and Y.

Examples. The sequence ABC is a common subsequence of both $X = \mathsf{ACBCD}$ and $Y = \mathsf{ABCBD}$.

The sequence ABCD is a longer (indeed the longest) common subsequence of X and Y (and so is ACBD).

In the *longest common subsequence problem* we are given[a] two sequences X and Y and wish to find a maximum-length common subsequence (or LCS) of both X and Y.

Exercises.

1. Design a solution for the given optimisation problem, i.e. define an algorithm which returns one of the largest common subsequences of the two given sequences X and Y.
2. Analyse the time complexity of your solution.

Would a greedy algorithm work?

[a]Notice that no explicit assumption about the basic alphabet is needed. If X and Y do not share a common alphabet LCS(X, Y) = 0.

Characterising a longest common subsequence

The LCS problem has an optimal-substructure property. A natural class of sub-problems to consider corresponds to pairs of prefixes of the two input sequences.

Let $X = x_1x_2 \dots x_m$, and $Y = y_1y_2 \dots y_n$ be sequences. and let $Z = z_1z_2 \dots z_k$ be any LCS of X and Y.

1. If $x_m = y_n$, then

   $z_k = x_m = y_n$ and Z_{k-1} is a LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then

   if $z_k \neq x_m$ then Z is a LCS of X_{m-1} and Y;

   if $z_k \neq y_n$ then Z is a LCS of X and Y_{n-1}.

Explanation

Let X and Y be two sequences as in the statement of the result and Z be an LCS of the two.

(Case 1.) Let's assume the two sequences X and Y end with the same character, but $z_k \neq x_m$. Then Zx_m is a common subsequence that is longer than Z (impossible). Therefore $z_k = x_m = y_n$. Now, Z_{k-1} is a common subsequence of length $k-1$ of X_{m-1} and Y_{n-1}. Suppose by contradiction that there exists a common subsequence W of X_{m-1} and Y_{n-1} with $|W| > k-1$. Then Wz_k is a common subsequence of X and Y which is longer than Z. Hence Z_{k-1} is one of the longest common subsequences of X_{m-1} and Y_{n-1}.

(Case 2.) Let's assume $z_k \neq x_m$ then Z is actually a common subsequence of X_{m-1} and Y. As above, if Z wasn't the LCS of X_{m-1} and Y the longest such sequence could be extended to be an LCS for X and Y. The case $z_k \neq y_n$ is symmetrical.

A recursive solution to subproblems

A recursive solution to the LCS problem also has the overlapping-subproblems property. The analysis so far implies that to find the LCS of X and Y we either have $x_m = y_n$, in which case the problem is reduced to compute the LCS of X_{m-1} and Y_{n-1} or else we find an LCS in X_{m-1} and Y or X and Y_{n-1}. Each of these subproblems has LCS(X_{m-1}, Y_{n-1}) as common sub-problem.

The recursive definition of the optimal cost is readily completed. If $lcs[i, j]$ is the longest common subsequence of X_i and Y_j, then

$$lcs[i,j] = \begin{cases} 0 & i = 0 \ \vee \ j = 0 \\ lcs[i-1, j-1] + 1 & i, j > 0 \ \wedge \ x_i = y_j \\ \max\{lcs[i, j-1], lcs[i-1, j]\} & i, j > 0 \ \wedge \ x_i \neq y_j \end{cases}$$

Computing the length of an LCS

One can easily write an exponential time recursive algorithm to compute lcs. However, since there are only $\Theta(nm)$ distinct subproblems, we can also use dynamic programming to compute the solutions bottom up.

The following procedure takes two sequences $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$ and stores the output in a two dimensional array lcs, whose entries are computed in row-major order.

The table $b[1..m, 1..n]$ will be used later to simplify the construction of an optimal solution.

On completion $lcs[m, n]$ contains the size of an LCS between X and Y.

The overall running time of the procedure is $O(nm)$ since each table entry takes $O(1)$ time to compute.

Constructing an LCS

The b table returned by the LCS procedure can be used quickly to construct an optimal solution for the given instance. We simply begin at $b[m, n]$ and trace through the table "following directions": the "$\nwarrow$" symbol implies that $x_i = y_j$ is an element of the LCS.


```
LCS (X, Y)
    m ← length(X)
    n ← length(Y)
    for i ← 1 to m lcs[i, 0] ← 0
    for j ← 0 to n lcs[0, j] ← 0
    for i ← 1 to m
        for j ← 1 to n
            if x_i = y_j
                lcs[i, j] ← lcs[i − 1, j − 1] + 1
                b[i, j] ← "↖"
            else if lcs[i − 1, j] ≥ lcs[i, j − 1]
                lcs[i, j] ← lcs[i − 1, j]
                b[i, j] ← "↑"
            else
                lcs[i, j] ← lcs[i, j − 1]
                b[i, j] ← "←"
    return lcs and b
```



```
PRINT-LCS (b, X, i, j)
    if i = 0 ∨ j = 0
        return
    if b[i, j] = "↖"
        PRINT-LCS (b, X, i − 1, j − 1)
        print x_i
    else if b[i, j] = "↑"
        PRINT-LCS (b, X, i − 1, j)
    else
        PRINT-LCS (b, X, i, j − 1)
```

The procedure takes $O(n + m)$ time, since at least one of i and j is decremented in each stage of the recursion.

Example

Let $X = 10010101$ and $Y = 010110110$.

To trace the algorithm execution it is convenient to follow the way in which the table lcs is filled. First of all, all entries in the first row and column are filled with zeroes. So, just before the third **for** loop in LCS is started , lcs will look like the table on the next page.

i	j	0	1	2	3	4	5	6	7	8	9
		y_i	0	1	0	1	1	0	1	1	0
0	x_i										
		0	0	0	0	0	0	0	0	0	0
1	1										
		0									
2	0										
		0									
3	0										
		0									
4	1										
		0									
5	0										
		0									
6	1										
		0									
7	0										
		0									
8	1										
		0									

Next we set i to one and j to one and we check $x_1 = 1$ against $y_1 = 0$. They are different, so we check $lcs[0,1] \geq lcs[1,0]$. These are both zero, hence the check succeeds. So $lcs[1,1]$ is set to $lcs[0,1]$ and $b[1,1]$ is set to "↑".

i	j	0	**1**	2	3	4	5	6	7	8	9
		y_i	0	1	0	1	1	0	1	1	0
0	x_i										
		0	0	0	0	0	0	0	0	0	0
1	1		↑								
		0	0								
2	0										
		0									
3	0										
		0									
4	1										
		0									
5	0										
		0									
6	1										
		0									
7	0										
		0									
8	1										
		0									

Next $j \leftarrow 2$ (we are comparing $X_1 \equiv 1$ with $Y_2 \equiv 01$). $x_1 = 1 = y_2$ hence $lcs[1,2] \leftarrow lcs[0,1] + 1$ and $b[1,2] \leftarrow$ "↖".

i	j	0	1	**2**	3	4	5	6	7	8	9
		y_i	0	1	0	1	1	0	1	1	0
0	x_i										
		0	0	0	0	0	0	0	0	0	0
1	1		↑	↖							
		0	0	1							
2	0										
		0									
3	0										
		0									
4	1										
		0									
5	0										
		0									
6	1										
		0									
7	0										
		0									
8	1										
		0									

Next $j \leftarrow 3$ (we are comparing $X_1 \equiv 1$ with $Y_2 \equiv 010$). $x_1 = 1$ but $y_3 = 0$. We then check $lcs[0,3] \geq\leftarrow lcs[1,2]$. The test gives a FALSE answer as $lcs[1,2] = 1$, hence such value is copied to $lcs[1,3]$ and and $b[1,2] \leftarrow$ "$\leftarrow$".

	j	0	1	2	**3**	4	5	6	7	8	9
i		y_i	0	1	0	1	1	0	1	1	0
0	x_i										
		0	0	0	0	0	0	0	0	0	0
1	1		↑	↖							
		0	0	1	← 1						
2	0										
		0									
3	0										
		0									
4	1										
		0									
5	0										
		0									
6	1										
		0									
7	0										
		0									
8	1										
		0									

Next $j \leftarrow 4$ (we are comparing $X_1 \equiv 1$ with $Y_2 \equiv 0101$). $x_1 = y_4$. So $lcs[1,4] \leftarrow lcs[0,3] + 1$ and $b[1,4] \leftarrow$ "$\nwarrow$".

	j	0	1	2	3	**4**	5	6	7	8	9
i		y_i	0	1	0	1	1	0	1	1	0
0	x_i										
		0	0	0	0	0	0	0	0	0	0
1	1		↑	↖		↖					
		0	0	1	← 1	1					
2	0										
		0									
3	0										
		0									
4	1										
		0									
5	0										
		0									
6	1										
		0									
7	0										
		0									
8	1										
		0									

The process should be clear now. Here is the final table.

	j	0	1	2	3	4	5	6	7	8	9
i		y_i	0	1	0	1	1	0	1	1	0
0	x_i										
		0	0	0	0	0	0	0	0	0	0
1	1		↑	↖		↖	↖		↖	↖	
		0	0	1	← 1	1	1	← 1	1	1	← 1
2	0		↖	↑	↖			↖			↖
		0	1	1	2	← 2	← 2	2	← 2	← 2	2
3	0		↖	↑	↖	↑	↑	↖			↖
		0	1	1	2	2	2	3	← 3	← 3	3
4	1		↑	↖	↑	↖	↖	↑	↖	↖	
		0	1	2	2	3	3	3	4	4	← 4
5	0		↖	↑	↖	↑	↑	↖	↑	↑	↖
		0	1	2	3	3	3	4	4	4	5
6	1		↑	↖	↑	↖	↖	↑	↖	↖	↑
		0	1	2	3	4	4	4	5	5	5
7	0		↖	↑	↖	↑	↑	↖	↑	↑	↖
		0	1	2	3	4	4	5	5	5	6
8	1		↑	↖	↑	↖	↖	↑	↖	↖	↑
		0	1	2	3	4	5	5	6	6	6

And here is the common subsequence (just pick the characters corresponding to a "$\nwarrow$" value of b starting from $b[m,n]$).

	j	0	1	**2**	**3**	4	5	**6**	**7**	**8**	**9**
i		y_i	0	**1**	**0**	1	1	**0**	**1**	**1**	**0**
0	x_i										
		0	0	0	0	0	0	0	0	0	0
1	**1**		↑	↖		↖	↖		↖	↖	
		0	0	1	← 1	1	1	← 1	1	1	← 1
2	**0**		↖	↑	↖			↖			↖
		0	1	1	2	← 2	← 2	2	← 2	← 2	2
3	**0**		↖	↑	↖	↑	↑	↖			↖
		0	1	1	2	2	2	3	← 3	← 3	3
4	**1**		↑	↖	↑	↖	↖	↑	↖	↖	
		0	1	2	2	3	3	3	4	4	← 4
5	0		↖	↑	↖	↑	↑	↖	↑	↑	↖
		0	1	2	3	3	3	4	4	4	5
6	**1**		↑	↖	↑	↖	↖	↑	↖	↖	↑
		0	1	2	3	4	4	4	5	5	5
7	**0**		↖	↑	↖	↑	↑	↖	↑	↑	↖
		0	1	2	3	4	4	5	5	5	6
8	1		↑	↖	↑	↖	↖	↑	↖	↖	↑
		0	1	2	3	4	5	5	6	6	6

Improvements

It is often the case that once you have developed an algorithm you find out that it is possible to improve it. Some improvements give no asymptotic improvement in the performance, others yield to substantial asymptotic savings in time and space.

- It is possible to eliminate the b table altogether. Each entry $lcs[i, j]$ depends on only three other values: $lcs[i-1, j-1]$, $lcs[i-1, j]$, and $lcs[i, j-1]$. Given $lcs[i, j]$ we can determine in $O(1)$ time which of the three values was used to compute it. Thus, we can reconstruct an LCS in $O(n+m)$ time using a procedure similar to PRINT-LCS (exercise!). Although we save $\Theta(mn)$ space by this method, the auxiliary space requirement for computing an LCS does not asymptotically decrease, since we still need $\Theta(mn)$ space for the table lcs.
- We can reduce the asyptotic space requirements since the main procedure to file lcf only needs two rows at a time: the one being

computed and the previous row. This improvement works if we only need lcs; if we need to reconstruct an optimal solution then the smaller table does not keep enough information to retrace our steps in $O(n+m)$ time.

- Currently David Eppstein, Zvi Galil, Raffaele Giancarlo and Giuseppe Italiano hold the record for the fastest LCS algorithm:

  Actually, if you look at the matrix above (MZ, this is lcs), *you can tell that it has a lot of structure – the numbers in the matrix form large blocks in which the value is constant, with only a small number of* corners *at which the value changes. It turns out that one can take advantage of these corners to speed up the computation. The current (theoretically) fastest algorithm for longest common subsequences (due to myself and co-authors) runs in time* $O(n \log |\mathcal{A}| + c \log \log \min(c, mn/c))$ *where* c *is the number of these corners, and* $\mathcal{A}$ *is the set of characters appearing in the two strings.*