Longest Common Subsequence

Subsequence. Given a string $S = s_1 s_2 \ldots s_n$, a subsequence of $S$ is any string $P = p_1 \ldots p_k$, such that:

1. For all $1 \leq i \leq k$, $p_i = s_j$ for some $j > 0$;
2. If $p_i$ is $s_j$ and $p_{i+1}$ is $s_l$, then $l > j$.

Informally, a subsequence $P$ of string $S$ can be obtained by removing zero or more characters from $S$ and preserving the order of the characters not removed.

Example. Let $S = ATCATTCGC$. Then, $ATC$, $AAT$, $ATATG$ and $CCCG$ are all subsequences of $S$, while $AAA$, $ATT$ and $CCT$ are not.

Longest Common Subsequence (LCS) Problem. The Longest Common Subsequence (LCS) problem is specified as follows: given two strings $S$ and $T$, find the longest string $P$ which is a substring for both $S$ and $T$.

Brute-Force Solution.

A na"ive algorithm for solving LCS is:

1. Enumerate all possible subsequences of $S$.
2. For each subsequence of $S$ check if it is also a subsequence of $T$.
3. Keep track of the longest common subsequence found and its length.

Analysis. A string $S = s_1 \ldots s_n$ has $2^n$ possible subsequences (each subsequence is essentially a choice of which characters are in and which characters are out). Some of these subsequences are not unique, but in a brute-force algorithm, there is no way to know that ahead of time. Checking if a string $T = t_1 \ldots t_m$ contains a subsequence $P = p_1 \ldots p_k$ can be done in $O(m + k) = O(m)$ (if $k > m$, the answer is an automatic "no") time. Thus, the overall complexity of the brute-force algorithm is $O(m2^n)$. 

Characterization of a Longest Common Subsequence

To help us develop an efficient algorithm for LCS, we need to be able to understand what a longest common subsequence of two sequences looks like. The following theorem provides the key idea for an efficient algorithm:

**Theorem.** Let $S = s_1 \ldots s_n$ and $T = t_1 \ldots t_m$ be two strings and let $P = p_1 \ldots p_k$ be their longest common subsequence. Then:

1. If $s_n = t_m$, then $p_1 \ldots p_{k-1}$ is the longest common subsequence of $s_1 \ldots s_{n-1}$ and $t_1 \ldots t_{m-1}$;
2. If $s_n \neq t_m$ and $p_k \neq s_n$, then $P$ is the longest common subsequence of $s_1 \ldots s_{n-1}$ and $T$.
3. If $s_n \neq t_m$ and $p_k \neq t_m$, then $P$ is the longest common subsequence of $S$ and $t_1 \ldots t_{m-1}$.

Given $S = s_1 \ldots s_n$ and $T = t_1 \ldots t_m$, let $c[i, j]$ (for $1 \leq i \leq n$ and $1 \leq j \leq m$) represent the length of the maximal longest subsequence of $s_1 \ldots s_i$ and $t_1 \ldots t_j$. For the sake of consistency we set $c[0, 0] = 0$.

The theorem suggests the following approach to determining the length of the LCS of $S$ and $T$:

- Build the matrix $c[i, j]$ from $c[0, 0]$ all the way to $c[n, m]$. $c[n, m]$ will contain the length of the LCS of $S$ and $T$.
- Make sure that the construction of the matrix allows for a fast determination of the actual LCS.

**Building the matrix** $c[i, j]$. Using the theorem above, we can derive the following about $c[i, j]$:

- If $s_i = t_j$ then $c[i, j] = c[i-1, j-1] + 1$.
  
  If the two last characters of the substrings agree, then the LCS extends to include this character.

- If $s_i \neq t_j$ then $c[i, j] = \max(c[i, j-1], c[i-1, j])$.
  
  Essentially, if the last characters of the substring differ, then the LCS of $s_1 \ldots s_i$ and $t_1 \ldots t_j$ is also the LCS of one of the two strings and the other string without the last character.

We represent this formally as the following recurrence relation:

$$c[i, j] = \begin{cases} 
0 & \text{if } i = j = 0; \\
\max(0, c[i-1, j-1] + 1) & \text{if } i, j > 0; s_i = t_j; \\
\max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0; s_i \neq t_j.
\end{cases}$$

Essentially, $c[i, j]$ can be determined if we know the values in the following cells: $c[i-1, j-1]$, $c[i, j-1]$ and $c[i-1, j]$. We can set $c[0, j] = 0$ and $c[i, 0] = 0$ for all $1 \leq j \leq m$ and $1 \leq i \leq n$. This makes it possible to compute $c[1, 1]$, which, in turn, makes it possible to compute $c[1, 2]$ and $c[2, 1]$, and so on.
"Remembering" the LCS. On each step \((i, j)\) of computation of \(c[i, j]\), we can determine which of the three cells \(c[i-1, j-1]\) (diagonally above and to the left), \(c[i, j-1]\) (to the left) or \(c[i-1, j]\) (above) is the one whose value is used in computing \(c[i, j]\).

We create a table \(u[i, j]\). In cell \(s[i, j]\) we store the "pointer" to the cell from which \(c[i, j]\) was constructed. We use symbols \(\leftarrow\), \(\uparrow\) and \(\downarrow\) to denote the following cases:

<table>
<thead>
<tr>
<th>(u[i, j])</th>
<th>Symbol</th>
<th>(s_i) vs. (t_j)</th>
<th>Table condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\downarrow)</td>
<td>(s_i = t_j)</td>
<td>(N/A)</td>
<td>(c[i-1, j-1] \geq c[i, j-1])</td>
</tr>
<tr>
<td>(\uparrow)</td>
<td>(s_i \neq t_j)</td>
<td>(c[i, j-1] &gt; c[i, j-1])</td>
<td>(c[i-1, j] &gt; c[i-1, j])</td>
</tr>
<tr>
<td>(\leftarrow)</td>
<td>(s_i \neq t_j)</td>
<td>(c[i-1, j] &gt; c[i, j-1])</td>
<td>(c[i-1, j] &gt; c[i-1, j])</td>
</tr>
</tbody>
</table>

Proposition. There is a path from \(s[n, m]\) to \(s[0, 0]\). The LCS of \(S = s_1 \ldots s_n\) and \(T = t_1 \ldots t_m\), given a constructed matrix \(u[i, j]\) can be found by combining all \(s_i\) characters for all locations \([i, j]\), where \(u[i, j] = \downarrow\).

Dynamic Programming Algorithm for LCS

To find the LCS of two strings, we need to construct the two matrices: \(c[i, j]\) and \(s[i, j]\). The following iterative version of the algorithm can do it.

Algorithm LCS\((S = s_1 \ldots s_n, T = t_1 \ldots t_m)\)

begin
  declare \(c[0..n, 0..m]\); 
  declare \(u[0..n, 0..m]\);
  for \(i = 0\) to \(n\) do 
    \(c[i, 0] := 0;\)
  end for
  for \(j = 1\) to \(m\) do
    \(c[0, j] := 0;\)
  end for
  for \(i = 1\) to \(n\) do
    for \(j = 1\) to \(m\) do
      if \(s_i = t_j\) then
        \(c[i, j] := c[i-1, j-1] + 1;\)
        \(u[i, j] := \downarrow;\)
      else
        if \(c[i-1, j] \geq c[i, j-1]\) then
          \(c[i, j] := c[i-1, j]\);
          \(u[i, j] := \uparrow;\)
        else
          \(c[i, j] := c[i, j-1]\);
          \(u[i, j] := \leftarrow;\)
        end if
      end if
    end for
  end for
  LCSLength := \(c[n, m]\);
  LCS := LCSRecover\((S, T, u[]);\)
  return \((LCS, LCSLength);\)
end
The algorithm LCSRecover takes as input two strings, $S$ and $T$, and the matrix $u[i,j]$ that encodes how $c[i,j]$ was filled, and returns back the LCS of $S$ and $T$. The algorithm works as follows (in the algorithm below, $+$ on string values is a concatenation operation).

```
Algorithm LCSRecover($S = s_1 \ldots s_n$, $T = t_1 \ldots t_n$, $u[0..n,0..m]$)
begin
    $P := ""$;
    $i := n$;
    $j := m$;
    $P := s_i + P$;
    while $i > 0$ and $j > 0$ do
        if $u[i,j] = $\ then
            $P := s_i + P$;
            $i := i - 1$;
            $j := j - 1$;
        else
            if $u[i,j] = \leftarrow$ then
                $j := j - 1$;
            else
                // $u[i,j] = \uparrow$
                $i := i - 1$;
            end if
        end if
    end while
    return $P$;
end
```

**Analysis.** Algorithm LCS contains a double nested loop that iterates $n \cdot m$ times. Each loop iteration completes in $O(1)$.

On each step of the main loop of the algorithm LCSRecover either $i$ or $j$ gets decreased (and on some steps, both $i$ and $j$ are decreased). This means that the main loop of LCSRecover runs no more than $m + n$ times, and the algorithm itself has $O(m + n)$ runtime complexity.

As a result, algorithm LCS has $O(nm) + O(n + m) = O(nm)$ runtime complexity.

---

1. It actually needs only one string, since it returns the common sequence.