An Efficient Algorithm for the LCS Problem
Longest Common Subsequence Problem

• The *longest common subsequence* problem, also called the *LCS* problem is a special case of the similarity problem.

• **Definition:** Given a string $S$ of length $n$, a *subsequence* is a string $S(i_1)S(i_2)\ldots S(i_k)$ such that $1 \leq i_1 < i_2 < i_3 \ldots < i_k$ for some $k \leq n$.

• A *substring* is a subset of $S$ which are located contiguously but in a subsequence the characters are not necessarily contiguous but they are in order from left to right.

• Thus a substring is a subsequence but the converse is not true.
Longest Common Subsequence

• **Definition:** The *longest common subsequence* or *LCS* of two strings $S_1$ and $S_2$ is the longest subsequence common between two strings.

$S_1: \ A \ -- \ A \ T \ -- \ G \ G \ C \ C \ -- \ A \ T \ A \ \ n=10$

$S_2: \ A \ T \ A \ T \ A \ A \ T \ T \ C \ T \ A \ T \ -- \ m=12$

The LCS is *AATCAT*. The length of the LCS is 6.

The solution is not unique for all pair of strings. Consider the pair ($ATTA$, $ATAT$). The solutions are $ATT$, $ATA$. In general, for arbitrary pair of strings, there may exist many solutions.
LCS Problem

• The LCS can be found by dynamic programming formulation. Since it is using the general dynamic programming algorithm its complexity is $O(nm)$.

• A longest substring problem, on the other hand has a $O(n+m)$ solution. Subsequences are much more complex than substrings.

• Can we do better for the LCS problem? We will see ...
LCS for $S_1$ and $S_2$

$S_1$: A -- A T -- G G C C -- A T A $n=10$
$S_2$: A T A T A A T T C T A T $m=12$

- The optimal alignment is shown above. Note the alignment shows three insert (dark), one delete (green) and three substitution or replacement operations (blue), which gives an edit distance of 7.
- But, the 3 replacement operations can be realized by 3 insert and 3 delete operations because a replacement is equivalent to first delete the character and then insert a character in its place like:

  G -- G -- C --
  -- A -- T -- T
Edit Distance and LCS are related

- if we give a cost of 2 for replace operation and cost of 1 for both insert and delete operations, the minimum edit distance $D$ can be computed in terms of the length $L$ of LCS as:

$$D = m + n - 2L$$

- For the above example, $n=10$, $m=12$, $L=6$. So, $D=10$ (6 insert and 4 delete).
Direct Computation of *LCS* by Dynamic Programming

• More efficient although the asymptotic complexity remains the same, $O(nm)$.

• Let $L$ denote The equations are given below without proof (which is simple).

$$L(0,0) = 0$$
$$L(i,0) = 0$$
$$L(0, j) = 0$$
$$L(i, j) = 1 + L(i-1, j-1)$$

$$S_1(i) = S_2(j)$$

• Again, if we leave suitable back pointers in the matrix, trace(s) can be derived for the *LCS*. 
Edit Graph for LCS Problem

$D = n + m - 2L$
$D = 7 + 8 - 2 \times 5 = 5$
A Faster Algorithm for LCS

• An algorithm that is asymptotically better than $O(nm)$ for determining LCS.
• Implies that for special cases of edit distance, there exist more efficient algorithm.
• Definition:
  – Let $\pi$ be a set of $n$ integers, not necessarily distinct.
• Definition:
  – An increasing subsequence (IS) of $\pi$ is a subsequence of $\pi$ whose values are strictly increasing from left to right.
• Example: $\pi=(5,3,4,4, 9,6,2,1,8,7,10)$. IS=(3,4,6,8,10), (5,9,10)
• Definition:
  – A *longest increasing subsequence (LIS)* of $\pi$ is an IS $\pi$ of maximum length.

• Definition:
  – A *decreasing subsequence (DS)* of $\pi$ is a non-increasing subsequence from left to right.

• Example: $DS = (5, 4, 4, 3, 2, 1)$. 
• Definition:
  – A cover is a set of disjoint DS of $\pi$ that covers or contains all elements of $\pi$. The size of the cover $c$ equals the number of DS in the cover.

• Example: $\pi=(5,3,4,9,6,2,1,8,7)$ Cover:{$(5,3,2,1),(4),(9,6),(8,7)$}. $c=$#of $DS=4$.

• Definition:
  – A smallest cover (SC) is a cover with a minimum value of $c$. 

Determine $\text{LIS}$ and $\text{SC}$ simultaneously in $O(n\log n)$

• **Lemma:**
  
  – If $I$ is an $\text{IS}$ of $\pi$ with length equal to the size of a cover $C$ of $\pi$, then $I$ is a $\text{LIS}$ of $\pi$ and $C$ is the smallest cover of size $c$. 
Proof

• If $I$ is an increasing sequence, it cannot contain more than one element from a decreasing sequence.

• This means that no increasing subsequence can have size more than the size of any cover $C$, that is, if

$$C = C_1 \cup C_2 \cup \ldots \cup C_c$$

a maximum of one element from each can participate in any increasing sequence.

• Thus, an IS derived from this decomposition can have a maximum length of $|C| = c$. Conversely, $C$ must be the smallest. If not, let $c'$ be the length of a cover $C'$ such that $|C'| = < c$. i.e., if we derive IS from $C$, it must contain more than one element from one of the decreasing sequence of $C'$, which is not possible. Hence $C$ has to be of smallest size.
Construction of a cover

• **Greedy algorithm** to derive a cover:
  – Starting from the left of $\pi$, examine each successive number in $\pi$.
  – Append the current number at the left-most subsequence derived so far if it is possible do that maintaining the decreasing sequence property.
  – If not start a new decreasing subsequence beginning with the current element.
  – Proceed until $\pi$ is exhausted.
Example

• \(\pi=(5,3,4,9,6,2,1,8,7,10)\)
• \(D1=(5,3,2,1),\; D2=(4),\; D3=(9,6),\; D4=(8,7),\; D4=(10)\)
• The algorithm has \(O(n^2)\) complexity. We will present an \(O(n \log n)\) algorithm.
An Efficient Algorithm for Constructing the Cover

• We use a data structure which is a list containing the last number of each of the decreasing sequence that is being constructed.

• The list is always sorted in increasing order. An identifier indicating which list the number belongs to also included.

Procedure Decreasing Sequence Cover

Input: $\pi=(x_1, x_2, ........x_n)$ the list of input numbers.
Output: the set of decreasing sequences $D_i$ constituting the cover.
O(n logn) Algorithm

- Initialize: \( i \leftarrow 1; \ Di = (x_1); \ L = (x_1, i) ; j \leftarrow 1; \)
- For \( i = 2 \) to \( n \) do
  - Search the \( x \)-fields of \( L \) to find the first \( x \)-value such that \( x_i < x \). ....takes \( O( \log n) \) time.
  - If such a value exists, then insert \( x \) at the end in the list \( Di \) and set \( x_i \leftarrow x \) in \( L \)... This step takes constant time.
  - If such a value does not exist in \( L \), then set \( j \leftarrow j+1 \). insert in \( L \) a new element \((x, j)\) and start a new decreasing sequence \( Dj = (x) \)

End
• Lemma:
  – At any point in the execution of the algorithm the list $L$ is sorted in increasing order with respect to $x$-values as well as with respect to identifier value.
• In fact two separate lists will be better from practical implementation point of view.
• Theorem:
  – The greedy cover can be constructed taking $O(n \log n)$ time. A longest increasing sequence and a smallest cover thus can be constructed using $O(n \log n)$ time.
**Example:** $\pi=(5,3,4,9,6,2,1,8,7,10)$

<table>
<thead>
<tr>
<th>$i$</th>
<th>$x_i$</th>
<th>$L$</th>
<th>$D_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>${5,1}$</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>${3,1}$</td>
<td>(5,3)</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>${3,1},(4,2)}$</td>
<td>(5,3)</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>${3,1},(4,2),(9,3)}$</td>
<td>(5,3)</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>${3,1},(4,2),(6,3)}$</td>
<td>(5,3)</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>${2,1},(4,2),(6,3)}$</td>
<td>(5,3,2)</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>${1,1},(4,2),(6,3)}$</td>
<td>(5,3,2,1)</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>${1,1},(4,2),(6,3),(8,4)}$</td>
<td>(5,3,2,1)</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
<td>${1,1},(4,2),(6,3),(7,4)}$</td>
<td>(5,3,2,1)</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>${1,1},(4,2),(6,3),(7,4),(10,5)}$</td>
<td>(5,3,2,1)</td>
</tr>
</tbody>
</table>

The $x$-component of the list, if separated, will look like the following during execution:

$(5),(3),(3,4), (3,4,9), (3,4,6), (2,4,6),(1,4,6), (1,4,6,8), (1,4,6,8),(1,4,6,7), (1,4,6,7,10)$
Reduction of \textit{LIS} problem to \textit{LCS} problem

• Definition:
  – Given sequences \( S_1 \) and \( S_2 \), let \( r(i) \) be the number of occurrence of the \( i \)th character of \( S_1 \) in \( S_2 \).

\[
\begin{align*}
\text{(position index in sequence } S_2:) & 1 2 3 4 5 6 \\
\text{Example:} S_1 & = a \ b \ a \ c \ x \text{ and } S_2 = b \ a \ a \ b \ c \ a \\
\text{Then, } r(1) & = 3, \ r(2) = 2, \ r(3) = 3, \ r(4) = 1, \ r(5) = 0 .
\end{align*}
\]
Definition: list(x)

• Definition:
  – for each distinct character $x$ in $S1$, define $list(x)$ to be the positions of $x$ in $S2$ in decreasing order.

• Example: $list(a) = (6,3,2); \ list(b) = (4,1),\ list(c) = (5), \ list(x) = \varnothing$ (empty sequence).
Definition: $\Pi (S_1,S_2)$

- Definition: Let $\Pi (S_1,S_2)$ be a sequence obtained by concatenating $\text{list}(s_i)$ for $i=1,2,\ldots,n$ where $n$ is the length of $S_1$ and $s_i$ is the $i$th symbol of $S_1$.
- Example: $\Pi (S_1,S_2)= (6,3,2,4,1,6,3,2,5)$. 
Theorem

• Theorem:
  – Every increasing sequence \( I \) of \( \Pi (S1,S2) \) specifies an equal length common subsequence of \( S1 \) and \( S2 \) and vice versa. Thus a longest common subsequence \( LCS \) of \( S1 \) and \( S2 \) corresponds to a longest increasing sequence of \( \Pi (S1,S2) \).

• Example: \( \Pi (S1,S2)= (6,3,2,4,1,6,3,2,5) \). The possible longest increasing sequences used as indices to access the characters in \( S2 \) yield the \( LCS \) as: \( (1,2,5)= b \ a \ c \), \( (2,3,5)=a \ a \ c \), \( (3,4,6)= a \ b \ a \) for \( S1= a \ b \ a \ c \ x \) and \( S2= b \ a \ a \ b \ c \ a \).