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 $S(i_1)S(i_2)....S(i_k)$ such that $1 \le i_1 < i_2 < i_3 \ldots < i_k$ for some $k \le n$.
- 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 *S1* and *S2* is the longest subsequence common between two strings.

S1 :	***A***	--	***A***	***T***	--	*G*	*G*	*C*	***C***	--	*A*	***T***	***A***	***n=10***
S2:	*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).

$$L(0,0) = 0$$

$$L(i,0) = 0$$

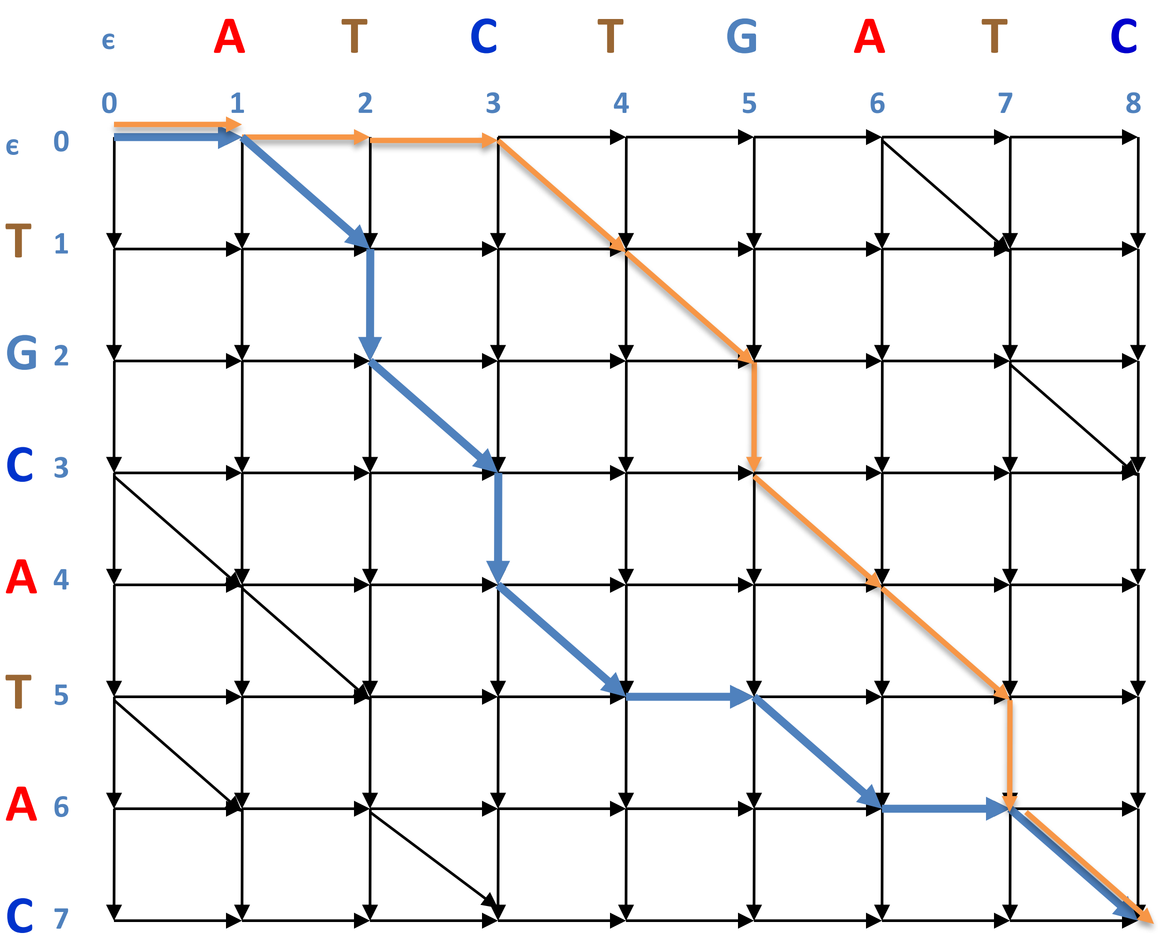
$$L(0, j) = 0$$

$$L(i, j) = 1 + L(i-1, j-1) \ldots\ldots\ldots\ldots\ldots\ldots\ldots S_1(i) = S_2(j)$$

$$L(i, j) = \max[L(i, j-1), L(i-1, j)] \ldots\ldots\ldots S_1(i) \neq S_2(j)$$

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

Edit Graph for LCS Problem



T G C A T A C
A T C T G A T C

- T G C A T - A – C
A T - C - T G A T C

T G C A T A C
A T C T G A T C

- - - T G C A T A C
A T C T G - A T - C

D=n+m-2L
D=7+8-2*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 π be a set of *n* integers, not necessarily distinct.
- Definition:
  - An *increasing* ***subsequence(IS)*** of π is a subsequence of π whose values are ***strictly*** increasing from left to right.
- Example: π=(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 π is an *IS* π of maximum length.
- Definition:
  - A ***decreasing subsequence (DS)*** of π is a non-increasing subsequence f π from left to right.
- Example: *DS*=(5,4,4,3,2,1).

- Definition:
  - A ***cover*** is a set of disjoint *DS* of π that covers or contains all elements of π. The size of the cover *c* equals the number of *DS* in the cover.
- Example: π=(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 *LIS* and *SC* simultaneously in $O(n\log n)$

- **Lemma**:
  - If *I* is an *IS* of π with length equal to the size of a cover C of π, then *I* is a *LIS* of π 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

$$C = C_1 \bigcup C_2 \bigcup ....\bigcup C_c$$

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 π, examine each successive number in π.
  - 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 π is exhausted.

Example

- π=(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 logn*)** 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: π=$(x_1, x_{2,} ....x_n,)$the list of input numbers.

Output: the set of decreasing sequences D_i constituting the cover.

$O(n \log n)$ Algorithm

- Initialize: $i \leftarrow 1$; $Di=(x1)$; $L=(x1, i)$; $j \leftarrow 1$;
- For $i=2$ to n do
  - Search the x-fields of L to find the *first* x-value such that $xi < x$. ....takes $O(\log n)$ time.
  - If such a value exists, then insert x at the end in the list Di and set $xi \leftarrow x$ 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: $\pi=(5,3,4,9,6,2,1,8,7,10)$

i=1	*x1*=5	*L*={(5,1)}	*D1*=(5)				
2	3	{(3,1)}	(5,3)				
3	4	{(3,1),(4,2)}	(5,3)	*D2*=(4)			
4	9	{(3,1),(4,2),(9,3)}	(5,3)	(4)	*D3*=(9)		
5	6	{(3,1),(4,2),(6,3)}	(5,3)	(4)	(9,6)		
6	2	{(2,1),(4,2),(6,3)	(5,3,2)	(4)	(9,6)		
7	1	{(1,1),(4,2),(6,3)}	(5,3,2,1)	(4)	(9,6)		
8	8	{(1,1),(4,2),(6,3),(8,4)}	(5,3,2,1)	(4)	(9,6)	*D4*=(8)	
9	7	{(1,1),(4,2),(6,3),(7,4)}	(5,3,2,1)	(4)	(9,6)	*D4*=(8,7)	
10	10	{(1,1),(4,2),(6,3),(7,4),(10,5)}	(5,3,2,1)	(4)	(9,6)	*D4*=(8,7)	*D5=(10)*

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 *LIS* problem to *LCS* problem

- Definition:
  - Given sequences *S1* and *S2*, let *r(i)* be the number of occurrence of the *i*th character of *S1* in *S2*.

(position index in sequence *S2:*) 1 2 3 4 5 6

Example:*S1=a b a c x* and *S2*= *b a a b c a*

Then, *r*(1)=3, *r*(2)=2, *r*(3)=3, *r*(4)=1, *r*(5)=0 .

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), *lis*t(*c*)=(5), *lis*t(*x*)=φ (empty sequence).

Definition: Π ($S1,S2$)

- Definition: Let Π ($S1,S2$) be a sequence obtained by concatenating ***list*(*si*)** for i=1,2,…,n where n is the length of $S1$ and si is the ith symbol of $S1$.
- Example: Π ($S1,S2$)= (6,3,2,4,1,6,3,2,5).

Theorem

- Theorem:
  - Every increasing sequence *I* of Π (*S1,S2*) specifies an equal length common subsequence of *S*1 and *S*2 and vice versa. Thus a longest common subsequence *LCS* of *S*1 and *S*2 corresponds to a longest increasing sequence of Π (*S1,S2*).
- Example: Π (*S1,S2*)= ($\underline{6,3,2}$,$\underline{4,1}$,$\underline{6,3,2}$,$\underline{5}$). 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.*